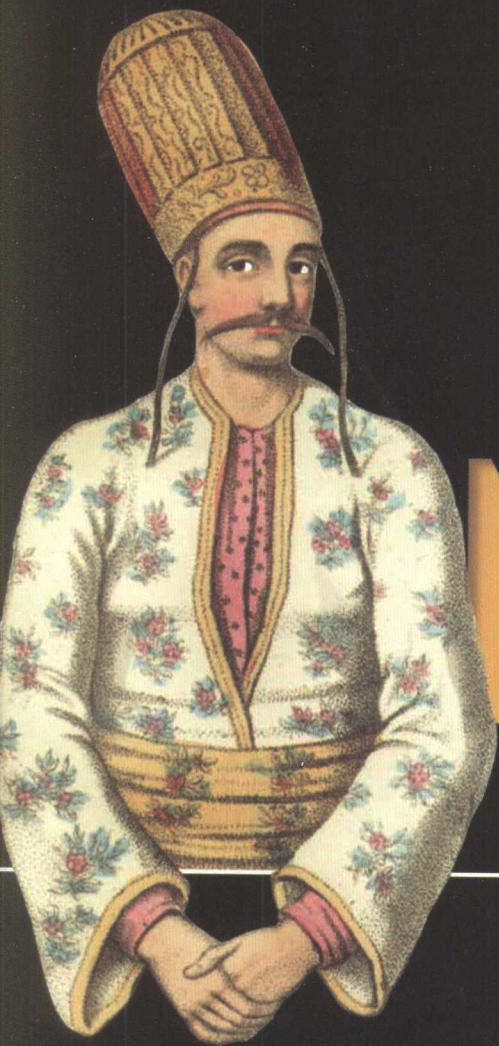


版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Ben Noordhuis作序



Node.js

硬实战

115个核心技巧

[美] Alex Young Marc Harter 著
承竹 慕陶 邱娟 达峰 译

Node.js in Practice

Node.js 硬实战 115个核心技巧

[美] Alex Young Marc Harter 著

承竹 慕陶 邱娟 达峰 译

Node.js in Practice

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

《Node.js 硬实战：115 个核心技巧》是一本面向实战的 Node.js 开发进阶指南。作为资深专家，本书作者独辟蹊径，将着眼点放在 Node.js 的核心模块和网络应用，通过精心组织的丰富实例，向读者充分展示了 Node.js 强大的并发处理能力，读者从中可真正掌握 Node 的核心基础与高级技巧。本书总共有三部分内容，第一部分是 Node.js 的基础核心，涉及 Buffer、流、网络 and 进程等相关知识；第二部分是项目实践，涉及测试、Web 开发、调试，生产环境等重要话题；第三部分则完整创建了一个 Node.js 模块。每部分涉及的技术都有详细讲解及注释详尽的示例代码，以帮助读者们更好地理解要点及其应用。

本书适合有一定 JavaScript 基础，追求在 Node.js 上更进一步的开发者。

Original English Language edition published by Manning Publications, USA. Copyright ©2015 by Manning Publications. Simplified Chinese-language edition copyright ©2016 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由 Manning Publications 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号图字：01-2015-3992

图书在版编目（CIP）数据

Node.js 硬实战：115 个核心技巧 / （美）亚历克斯·荣（Alex R. Young），（美）马克·哈特（Marc Harter）著；承竹等译.—北京：电子工业出版社，2017.1

书名原文：Node.js in Practice

ISBN 978-7-121-30402-6

I. ① N…II. ① 亚… ② 马… ③ 承…III. ① JAVA 语言—程序设计 IV. ① TP312.8

中国版本图书馆 CIP 数据核字（2016）第 277756 号

责任编辑：张春雨

印 刷：三河市良远印务有限公司

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：30

字数：672 千字

版 次：2017 年 1 月第 1 版

印 次：2017 年 1 月第 1 次印刷

定 价：109.90 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819 faq@phei.com.cn。

译者序

JavaScript 从它诞生以来就在浏览器应用中发挥着愈来愈重要的作用，同时，热爱 JavaScript 的人们一直在努力地将其应用在服务器领域。2009 年，Ryan Dahl 带来了 Node.js，从那个时候开始，JavaScript 社区出现了前所未有的繁荣。现在，Node.js 对开发者来说，几乎已经是家喻户晓的了。对于传统的服务器端开发者们而言，Node.js 带来了浏览器端使用已久的异步编程相关的概念，而对于前端开发者们来说，Node.js 则是带来了编写服务端程序方面的挑战。当我们这些对 Node.js 一知半解的开发者野心勃勃想要征服 Node.js 时，我们需要一个经验满满的导师来指引我们前行。

我很幸运地有这样一个机会接触到这本书，并且承担起翻译的任务。我第一次阅读原著时就感觉，它是一本可以胜任 Node.js 导师这个角色的书籍。作者认真负责地介绍了 Node.js 相关的方方面面，并且附带了相当详细的例子来帮助读者快速地理解其中的要点。我相信，对于想要学习 Node.js 的开发者来说，本书会是一个相当好的选择。

翻译书的过程，也是一个学习的过程，原谅我们水平有限，书中翻译内容难免有疏漏。当这本书差不多翻译完成的时候，我发现已经用了好长的时间，Node.js 技术更新得相当快，我相信小部分内容的细节上和现在的应用可能会出现些许差异，还请读者们谅解。

最后，在此对支持本书翻译工作的所有人们表示感谢，尤其是耐心容忍我们一再推迟交稿的编辑，真的非常感谢。

衷心希望本书能够对您有所帮助，请享受 Node.js 给您带来的一切，谢谢。

序

在你手中的是一本将带你进入了解 Node 旅程的书。在接下来看到的页面中，Alex Young 和 Mare Harter 会帮助你深刻地把握 Node 的核心模块、网络应用等。

网络应用，是 Node.js 光芒闪耀的地方。亲爱的读者，你可能已经很清楚这一点，我敢说，这就是你购买本书的原因！对于读了序的这少部分人，让我来告诉你们，这一切是如何开始的。

在一开始的时候，有一个 C10K 的问题，这个问题是这样的：如果你要在现代的硬件中处理 10000 个并发连接，你会怎么做？

你可以看到，操作系统处理大量的网络连接在持续很长一段时间里都是很糟糕的。硬件在很多方面很糟糕，软件在另外一些方面也很糟糕，当硬件和软件集成在一起的时候……语言学家暂时还没有合适的词语，单纯用糟糕来形容对这一切并不公平。幸运的是，技术是一个进步的故事，硬件越来越好，软件越来越智能。操作系统，如用户软件等，在管理大量的网络连接上有了很大的进步。

在很久以前，我们征服了 C10K 问题，现在目标转移了，我们已经把眼光投向了 C100K、C500K、C1M 问题。一旦我们轻松地跨越这些界限时，我完全相信，C10M 会是下一个问题。

Node.js 是这个并发性不断增长的故事的一部分，它的未来是光明的。我们生活在一个日益互联的世界，这个世界需要一个强大的工具来连接一切。我相信 Node.js 会是那个强大的工具，我希望，在读完这本书时，你会有同样的感受。

Ben Noordhuis
Cofounder, StrongLoop, Inc.

前言

当 Node 在 2009 年出现的时候，我们知道有一些东西不一样了。在服务端 JavaScript 并不是什么新鲜的东西。事实上，服务端的 JavaScript 几乎和客户端的 JavaScript 存在的时间一样长。Node 中，JavaScript 运行时的速度，再加上基于事件的并行，这些很多熟悉 JavaScript 程序员都熟悉的东西，确实是让人感到不可抗拒。不仅仅是像我们这样背景的客户端 JavaScript 开发者——Node 吸引了从系统层面到各种服务端开发、PHP、Ruby 或者 Java 的开发者们。我们都可以发现自己身处于这变化中。

在那个时候，Node 有很多变化，我们受困于它，但是在这个过程中也学习了很多东西。从一开始，Node 关注于一个小的、低级别的核心库，来为大量多样化并且增长的用户提供足够的功能。值得庆幸的是，因为早期的一些设计决定，今天大量多样化的用户空间还存在着。Node 现在更加稳定，并且在许多初创企业和已成功企业生产环境中使用。

当 Manning 联系我们来编写一本中级的、关于 Node 的书时，我们想到了在过去和常见陷阱做斗争时，以及在 Node 社区中获得的经验教训。尽管我们非常喜欢那些提供给开发人员的大量真正优秀的第三方模块，但是，我们留意到越来越少的开发人员接触到 Node 核心基础的教学。所以我们着手编写了《Node.js 硬实战》，来以一种深入和彻底的方式，探索 Node 的根源和基础，并且解决很多我们个人遇见过的，或者看到他人处理过的问题。

致谢

我们要感谢很多人，没有他们的帮助和支持，这本书不可能完成。

感谢 Manning Early Access Program (MEAP) 的在作者在线论坛发布评论和纠正的读者们。

感谢那些在书的各个阶段提供了宝贵反馈意见的技术评审们：Alex Garrett、Brian Falk、Chris Joakim、Christoph Walcher、Daniel Bretoi、Dominic Pettifer、Dylan Scott、Fernando Monteiro Kobayashi、Gavin Whyte、Gregor Zurowski、Haytham Samad、JT Marshall、Kevin Baister、Luis Gutierrez、Michael Piscatello、Philippe Charrière、Rock Lee、Shiju Varghese 和 Todd Williams。

感谢帮助我们每一步的 Manning 团队，尤其是我们的开发编辑 Cynthia Kane、编辑 Benjamin Berg、校对 Katie Tennant，以及其他在幕后工作的人们。

特别感谢 Ben Noordhuis 为本书写序，还有 Valentin Crettaz 和 Michael Levin 在书出版前仔细的技术校对。

Alex Young

如果没有 DailyJS 社区的鼓励，我没法完成这样的一本书。感谢近几年来和我分享模块和类库的每一个人：没有你们就无法跟上 Node.js 社区的脚步。同时也要感谢我的同事允许让我在生产环境中使用 Node.js。最后，感谢 Yuka 让我相信我自己可以从事创业和写书这些如此疯狂的事情。

Marc Harter

我要感谢 Ben Noordhuis、Isaac Schlueter，还有 Timothy Fontaine，为了他们关于 Node 的各种 IRC 讨论：你们了解底层系统，以一种深刻的方式支持和影响着 Node，从你们身上我们获益匪浅。另外，我要感谢和我一起写书的 Alex，似乎很难能有像我这样和 Alex 一起写书的机会，更加有趣的是他是一个在美国中西部的使用英语的人。最后我要感谢的是我的妻子，她让这一切变得可能，我由衷地说一句，你是如此可爱，谢谢你。

关于本书

《Node.js 硬实战》提供给读者来深入了解 Node 的核心模块和包系统。我们相信这是成为一个多产和自信的 Node 开发者的基础。不幸的是，因为巨大并且充满活力的第三方生态系统几乎为所有的任务提供了预置好的模块，所以小小的核心很容易被错过。在这本书中，我们在官方文档的基础上来进一步实践和深入。我们想要读者能够仔细分析和研究他们编写的项目以及项目所包含的第三方模块的内部工作。

这本书不是一本 Node 的入门级别的读物。入门的话，我们建议阅读 Manning 的《Node 实战》(*Node.js In Action*)。这本书的受众是那些对 Node 有一定经验，正在寻求更进一步的读者。建议有一定的 JavaScript 基础的读者，最好也熟悉 Windows、OS X 或者 Linux 命令行的读者阅读。

同时，我们注意到很多 Node 开发者有来自客户端的 JavaScript 开发背景。因此，我们会花一些时间来解释一些不大熟悉的概念，如处理二进制数据、底层网络和文件系统的工作，以及和主机操作系统进行交互——所有这些都使用 Node 来作为教学指南。

章节路线图

这本书共分为三部分。

第一部分包括了 Node 的核心基础，我们关注于那些可能用到的 Node 核心模块（非第三方模块）。第 1 章简要概述了 Node 的目的和意义。第 2 到第 8 章每一章内容会深入 Node 核心的不同方面，从 Buffers 到流，从网络到子进程。

第二部分内容关注于真实世界的开发技巧。第 9 到第 12 章的内容，将帮助你掌握 4 种

非常实用的技能——测试、Web 开发、调试以及在生产环境运行 Node。除了 Node 核心模块之外，这些章节的内容也会包括许多第三方模块的使用。

第三部分将指引你以一种直接的方式来创建自己的 Node 模块，使用 npm 命令的各种方法来处理打包、运行、测试、基准测试和共享模块。它同时也包括进行有效的项目版本化的有用提示。

整本书有 115 个技巧，每一部分包括了一个特定的 Node.js 主题或者任务，包括了实际问题、解决方案和讨论部分。

代码约定和下载

在本书中的所有代码都是像这样的固定宽度的字体，如 `fixed-width font like this`，以和周边其他文本内容区分开来。在很多列表中，代码中的关键概念会有相应的注解，带有编号的符号有时候可以帮助在文本中找到其他额外关于代码的有用信息。

这本书的代码风格基于 Google JavaScript 的编码风格¹。这意味着我们把 `var` 声明独立开来，使用驼峰式来命名函数和变量，并且通常都带分号。我们的风格是在 Node 社区使用的多种 JavaScript 代码风格的混合。

大部分书中所示的代码可以以不同的形式在伴随着书的示例源代码中找到。示例源代码可以免费地从博文视点网站上下载：www.broadview.com.cn，也可以在 Github 上关注：<https://github.com/alexyoung/nodeinpractice>。

读者在线服务

购买《Node.js 硬实战》并注册账号，你可以在上边对本书进行评论、提交勘误或者咨询问题，从编辑或者其他用户那里获得帮助。

¹<https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml>

关于封面插图

《Node.js 硬实战》的封面插图的标题是“来自 Ayvalik 的年轻人”，Ayvalik 是在土耳其爱琴海岸的一个小镇。这个插图是从奥斯曼帝国在 1802 年 1 月 1 日发布的伦敦 Old Bond 街的 William Miller 创作的一系列服饰中获取的。标题页在整个集合中丢失了，我们已经无法追溯其日期。在书的内容表格中带有英文和法文标识，每个插图都有两位艺术家的名字。两百年之后，你一定会惊讶地发现，他们的艺术作品竟让计算机编程的书变得如此优雅、有魅力。

这一图片系列是由 Manning 的一位编辑在 Manhattan 的西第二十六街 Garage 的一个跳蚤市场购买的。卖家是一个在土耳其 Ankara 的美国人，交易就发生在他正要收摊的时候。Manning 的编辑身上并没有足够的现金，信用卡和支票都被礼貌地拒绝了。当天晚上卖家就要坐飞机回 Ankara，好像没有希望了。结果是怎么解决的？仅仅是握手，一个老式的 verbal 协议。卖家简单地提出将钱电汇给他就好，编辑带着一张写了银行信息的纸和胳膊下的公事包就走了。当然，第二天我们就把钱汇过去了。我们依旧很感谢这个陌生人对我们的信任。这是很久前的一段回忆。

在 Manning 的我们，为自己的创意和突破，以及基于带回来的两个世纪前的照片集的计算机书籍封面能如此丰富多彩而庆幸。

目录

第一部分 Node 基础

1 入门.....	2
1.1 Node 入门.....	3
1.1.1 为什么使用 Node.....	3
1.1.2 Node 的主要特性.....	5
1.2 构建一个 Node 应用.....	7
1.2.1 创建一个新的 Node 项目.....	8
1.2.2 创建一个流的类.....	9
1.2.3 使用流.....	10
1.2.4 编写测试.....	11
1.3 总结.....	13
2 全局变量: Node 环境.....	15
2.1 模块.....	16
技巧 1 安装与加载模块.....	16
技巧 2 创建与管理模块.....	17
技巧 3 加载一组相关的模块.....	20
技巧 4 使用路径.....	22
2.2 标准 I/O 以及 console 对象.....	23
技巧 5 标准 I/O 流的读写.....	24

技巧 6	打印日志消息	25
技巧 7	基准测试	27
2.3	操作系统与命令行	29
技巧 8	获取平台信息	29
技巧 9	传递命令行参数	30
技巧 10	退出程序	31
技巧 11	响应信号量	33
2.4	使用 timer 延迟执行	35
技巧 12	通过 setTimeout 延迟执行函数	35
技巧 13	通过定时器定时调用回调函数	37
技巧 14	安全的操作异步接口	38
2.5	总结	41
3	Buffers: 使用比特、字节以及编码	43
3.1	修改数据编码	44
技巧 15	Buffer 转换为其他格式	44
技巧 16	使用 Buffers 来修改字符串编码	46
3.2	二进制文件转换为 JSON	49
技巧 17	使用 Buffer 来转换原始数据	49
3.3	创建你自己的二进制协议	65
技巧 18	创建自己的网络协议	65
3.4	总结	71
4	Events: 玩转 EventEmitter	72
4.1	基础用法	73
技巧 19	从 EventEmitter 继承	73
技巧 20	混合 EventEmitter	76
4.2	异常处理	78
技巧 21	管理异常	78
技巧 22	通过 domains 管理异常	80

4.3	高级模式	82
	技巧 23 反射	82
	技巧 24 探索 EventEmitter	85
	技巧 25 组织事件名称	87
4.4	第三方模块以及扩展	88
	技巧 26 EventEmitter 的替代方案	89
4.5	总结	91
5	流：最强大和最容易误解的功能	93
5.1	流的介绍	94
	5.1.1 流的类型	94
	5.1.2 什么时候使用流	94
	5.1.3 历史	95
	5.1.4 第三方模块中的流	96
	5.1.5 流继承事件	97
5.2	内置流	98
	技巧 27 使用内置的流来实现静态 web 服务器	98
	技巧 28 流的错误处理	101
5.3	第三方模块和流	102
	技巧 29 使用流的第三方模块	102
5.4	使用流基类	105
	技巧 30 正确地 from 流的基类继承	105
	技巧 31 实现一个可读流	107
	技巧 32 实现一个可写流	111
	技巧 33 使用双工流转换和接收数据	113
	技巧 34 使用转换流解析数据	114
5.5	高级模式和优化	118
	技巧 35 流的优化	118
	技巧 36 使用老的流 API	121
	技巧 37 基于功能的流适配	123
	技巧 38 测试流	125

5.6 总结	128
6 文件系统：通过异步和同步的方法处理文件.....	129
6.1 fs 模块概述	130
6.1.1 POSIX 文件系统包装器	130
6.1.2 流	132
6.1.3 批量文件操作	133
6.1.4 文件监视	133
6.1.5 同步的替代方案	133
技巧 39 读取配置文件	134
技巧 40 使用文件描述	136
技巧 41 使用文件锁	137
技巧 42 递归文件操作	142
技巧 43 编写文件数据库	147
技巧 44 监视文件以及文件夹	151
6.2 总结	154
7 网络：Node 真正的“Hello, World”	156
7.1 Node 中的网络	156
7.1.1 网络技术	157
7.1.2 Node 网络模块	161
7.1.3 非阻塞网络和线程池	162
7.2 TCP 客户端和服务端	163
技巧 45 创建 TCP 服务端和客户端	163
技巧 46 使用客户端测试 TCP 服务端	165
技巧 47 改进实时性低的应用	168
7.3 UDP 客户端和服务端	170
技巧 48 通过 UDP 传输文件	170
技巧 49 UDP 客户端服务应用	174
7.4 HTTP 客户端和服务端	179
技巧 50 HTTP 服务器	179

技巧 51 重定向	181
技巧 52 HTTP 代理	186
7.5 创建 DNS 请求	189
技巧 53 创建 DNS 请求	189
7.6 加密	191
技巧 54 一个加密的 TCP 服务器	192
技巧 55 加密的 Web 服务器和客户端	196
7.7 总结	198
8 子进程：利用 Node 整合外部应用程序	200
8.1 执行外部应用程序	202
技巧 56 执行外部应用程序	202
8.1.1 路径和 Path 的环境变量	203
8.1.2 执行外部程序时候出现的异常	204
技巧 57 流和外部应用程序	205
8.1.3 外部应用程序的串联调用	206
技巧 58 在 shell 中执行命令	208
8.1.4 安全性和 shell 命令执行	209
技巧 59 分离子进程	210
8.1.5 父进程和子进程之间的 I/O 处理	211
8.1.6 引用计数和子进程	213
8.2 执行 Node 程序	213
技巧 60 执行 Node 程序	214
技巧 61 Forking Node 模块	216
技巧 62 运行作业	218
8.2.1 工作池	220
8.2.2 使用池模块	222
8.3 同步运行	223
技巧 63 同步子进程	223
8.4 总结	227

第二部分 实践中的技巧

9 网络：构建精简的网络应用.....	230
9.1 前端技术	231
技巧 64 快速的静态网站服务器	231
技巧 65 在 Node 中使用 DOM	236
技巧 66 在浏览器端使用 Node 模块	238
9.2 服务端技术	241
技巧 67 Express 路由分离	241
技巧 68 自动重启服务器	245
技巧 69 配置 web 应用	248
技巧 70 优雅地处理错误	253
技巧 71 RESTful web 应用	257
技巧 72 使用自定义的中间件	267
技巧 73 使用事件进行解耦	273
技巧 74 使用 WebSockets 来处理 sessions	276
技巧 75 升级 Express 3 到 4	281
9.3 web 应用程序的测试	285
技巧 76 测试路由	286
技巧 77 为中间件注入创建 seams	288
技巧 78 测试依赖远程服务的应用	291
9.4 全栈框架	297
9.5 实时服务	299
9.6 总结	300
10 测试：编写健壮代码的关键.....	301
10.1 Node 测试的相关介绍	303
10.2 使用断言编写简单的测试	304
技巧 79 用内置的模块编写测试	305
技巧 80 编写验证异常的测试	308
技巧 81 创建自定义的断言	312

10.3 测试装置	314
技巧 82 使用一个测试装置组织测试	314
10.4 测试框架	318
技巧 83 使用 Mocha 编写测试	319
技巧 84 使用 Mocha 测试 web 应用	323
技巧 85 万能测试协议 (TAP)	328
10.5 测试工具	331
技巧 86 持续集成	331
技巧 87 数据库装置	335
10.6 扩展阅读	343
10.7 总结	343
11 调试：用于发现和解决问题.....	344
11.1 内省	345
11.1.1 显式异常	345
11.1.2 隐藏的异常	346
11.1.3 错误事件	346
11.1.4 错误参数	347
技巧 88 处理未捕获的异常	348
技巧 89 检查我们的 Node 代码	351
11.2 问题的调试	352
技巧 90 使用 Node 内置的调试器	352
技巧 91 使用 Node Inspector	359
技巧 92 对 Node 应用进行性能分析	361
技巧 93 内存泄漏的调试	365
技巧 94 使用 REPL 来检测运行中的程序	370
技巧 95 跟踪系统调用	377
11.3 总结	381

12 生产环境中的 Node: 安全地部署应用程序	382
12.1 部署	383
技巧 96 将 Node 程序部署到云端	383
技巧 97 使用 Apache 和 Ngnix 部署 Node 程序	389
技巧 98 在 80 端口上安全地运行 Node 程序	392
技巧 99 保持 Node 进程一直运行	394
技巧 100 在生产环境中使用 WebSockets	396
12.2 Node 程序的缓存和扩展性	402
技巧 101 HTTP 缓存	402
技巧 102 为程序的路由和扩展使用 Node 代理	404
技巧 103 使用集群保持程序的扩展性和弹性	408
12.3 维护	413
技巧 104 包的优化	413
技巧 105 日志和日志服务	415
12.4 更多关于 Node 程序的扩展性和弹性的备注	418
12.5 总结	419

第三部分 编写模块

13 编写模块, 掌握 Node 的所有	422
13.1 头脑风暴	424
13.1.1 更快的斐波那契模块	424
技巧 106 计划编写我们的模块	425
技巧 107 验证我们模块的想法	427
13.2 创建 package.json 文件	433
技巧 108 创建 package.json 文件	433
技巧 109 依赖处理	436
技巧 110 语义化版本号	441
13.3 用户体验	444
技巧 111 添加可执行脚本	444
技巧 112 在本地测试模块	446

技巧 113 在不同版本 Node 中测试	448
13.4 发布	451
技巧 114 发布模块	451
技巧 115 使用私有模块	453
13.5 总结	455
A 社区.....	457

第一部分

Node 基础

Node 拥有极小的标准库来为模块开发人员提供最底层的 API。虽然找到第三方模块相对容易，但很多任务可以不用它们来完成。在接下来的章节中我们将深入了解一些核心模块并且探索如何在实际中使用它们。

通过强化你对这些模块的理解，你将成为一个更为全面的 Node 开发者。在分析第三方模块时，你也能拥有更多的自信和理解。

1 入门

本章概要

- 为什么使用 Node
- Node 的主要特性
- 构建一个 Node 应用

Node 已经迅速成为一个可行并且真正高效的 web 开发平台。在 Node 诞生之前，在服务端运行 JavaScript 是件不可思议的事情，并且对其他的脚本语言来说，要实现非阻塞 I/O 通常需要依赖特殊的类库。但 Node 的出现改变了这一切。

JavaScript 与非阻塞 I/O 的组合极为强大：在 JavaScript 与生俱来的 callback 特性下，我们能在同一进程中异步地操作文件读写、网络 sockets 以及其他的 I/O 操作。

这本书面向的是有一定经验的 Node 开发者，因此本章只是帮助你进行快速的复习。如果你想从 Node 基础开始系统的学习，你可以阅读 *Node.js in Action*（作者 Mike Cantelon、Marc Harter、TJ Holowaychuk，以及 Nathan Rajlich；Manning 出版社 2013 年出版）。

在这一章中，我们将介绍 Node 是什么，它的工作原理，以及为什么它是不可或缺的。在第 2 章你将从 Node 全局对象——那些所有的 Node 进程都能访问的对象与方法开始体验 Node。

阅读之前

本书面向的是有一定经验的中高级 Node 开发者，虽然本章内容涵盖了一些入门资料，但后面的章节难度会很大。建议 Node 的入门开发者在阅读本书之前先看一下 *Node.js in Action*。

1.1 Node 入门

Node 是一个针对网络应用开发的平台，它基于 Google 的 JavaScript 运行时引擎 V8。但它不仅仅只是 V8。Node 的标准类库是它非常重要的一部分。它涵盖了从 TCP 服务端到同步或者异步的文件管理。这本书将教你如何正确地使用这些模块。

在这之前，让我们从一些场景入手来看看为什么使用 Node 以及何时应该使用它。

1.1.1 为什么使用 Node

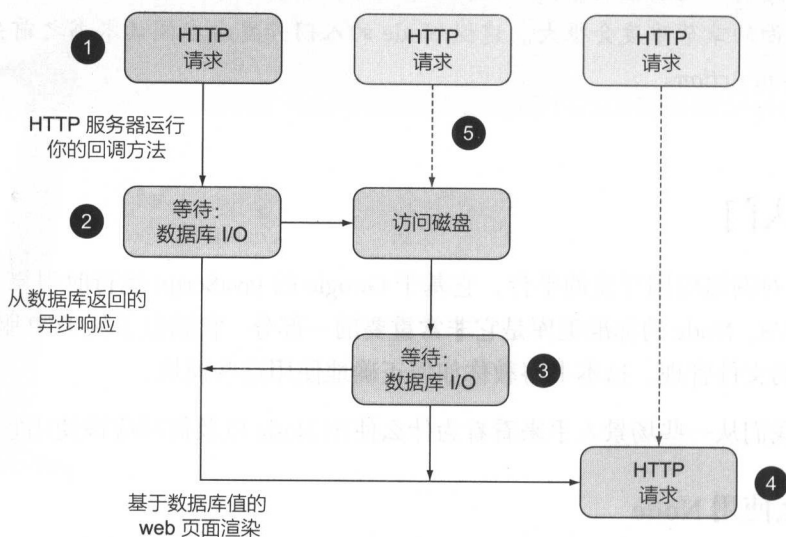
假设你正在开发一个广告服务器，每分钟需要发布几百万条的广告。Node 的非阻塞 I/O 将是一个高效的解决方案，因为服务器能够最大限度地利用到所有的 I/O 资源，而这一切不需要你写特殊的底层代码。并且，假如你已经有一支会写 JavaScript 的开发团队，那么他们应该可以直接参与到 Node 的项目中。传统的 web 平台将无法做到这一点，这也是为什么像微软这样的公司也在积极地推动 Node，尽管他们已经有了像 .NET 那么优秀的平台。Visual Studio(.NET IDE) 的用户可以安装一些工具¹来支持对 Node 的智能提示、性能监测，甚至 npm。微软还开发了 WebMatrix，它不但能直接支持 Node，还能部署 Node 项目。

Node 把非阻塞 I/O 作为提高某些类型应用的性能的方式。JavaScript 传统的事件机制意味着在异步编程中，它有着相对方便以及容易理解的语法。在传统的编程语言中，I/O 的操作将阻塞进程直到它完成为止。Node 的异步文件读写以及网络 API 意味着在这些相对较慢的 I/O 操作处理的时候主进程仍然能处理其他请求。图 1.1 展示了如何使用异步的网络和文件 API 同时处理多个任务。

在图 1.1 中，Node 的 http 模块接收到并且解析了一个新的 HTTP 请求①，然后服务端的应用代码使用异步接口，将一个回调函数传入数据库的读取函数中来进行一次数据查询②。在等待数据返回的同时服务器能够从文件系统中读取网页模板文件③，这个模板

¹ 参见 <https://nodejstools.codeplex.com/>。

文件被用来展示网页。一旦数据库完成查询，模板内容和数据库的返回数据将被用来渲染页面^④。



- ① 接收一个从浏览器来的 HTTP 请求。
- ② Node 解析请求后，你的代码会运行一个数据库查询。
- ③ 当查询回调方法在等待运行时，其他的代码会读取一个 HTML 模板文件。
- ④ 然后 Web 页面会基于模板文件和数据库的值来进行渲染。
- ⑤ 同时，其他请求同样可以被处理。

图 1.1 用 Node 构建的广告服务器

在服务器处理这个请求的同时，服务器还可以用可用的资源处理其他的请求^⑤。在不考虑多线程的情况下开发这个广告服务，你可以仅使用最基本的 JavaScript 编程技术，通过 Node，非常高效地使用服务器 I/O 资源。

其他 Node 适用的场景是 Web API 和网络爬虫，如果你需要下载以及截取网页的内容，那么 Node 将是非常完美的解决方案，因为它能模拟 DOM 操作，并且运行客户端 JavaScript 脚本。而且在这个场景中，Node 有着性能优势，因为网络爬虫主要的消耗在于网络 and 文件读写的 I/O。

假如你需要调用或者开发一个 JSON API，Node 也是一个非常棒的选择，因为它使得操作 JavaScript 对象变得非常简单。Node 的一些 web 框架（例如 express）能够快速搭建 JSON API，我们将在第 9 章详细讨论这个。

Node 不仅仅局限于 web 应用，你可以创建任意你想要的 TCP/IP 服务，比如网络游戏服务器，通过 TCP/IP 套接字向各类玩家发送游戏状态，也可以在后台任务中维护游戏数据，将数据发送给玩家。在第 7 章中，我们将探索 Node 的网络 API。

什么时候使用 Node

下面的表格有一些 Node 适用的应用例子，来帮你像一个真正的 Node 开发者一样来考虑这个问题。

Node 的强项

情景	Node 的强项
广告分布系统	<ul style="list-style-type: none">• 有效地分配小块信息• 处理潜在的网络速度慢的连接• 容易扩展为多个处理器或服务器
游戏服务器	<ul style="list-style-type: none">• 使用 JavaScript 来构建业务逻辑模型• 不使用 C 语言来开发迎合特定网络的服务器程序
内容管理系统、博客	<ul style="list-style-type: none">• 对已经有客户端 JavaScript 开发经验的团队来说，可以很轻松地创建 RESTful JSON APIs• 轻量的服务器，和浏览器端 JavaScript 结合

1.1.2 Node 的主要特性

Node 的主要特性是它的标准类库、模块系统以及 npm（包管理系统），当然还有很多其他的。但本书主要针对教你如何使用 Node 的这些部分。我们将使用被认为是最佳实践的第三方类库，但你会看到很多 Node 的内置特性。

实际上 Node 最强大的特性是它的标准类库，它主要由二进制类库以及核心模块两部分组成，二进制类库包括 libuv，它为网络以及文件系统提供了快速的事件轮循以及非阻塞的 I/O。同时它还有 http 类库，所以你可以很快确定你的 http 客户端与服务端。

图 1.2 是对 Node 内部的高层次概述，展示了各个模块是如何组合的。

Node 的核心模块主要由 JavaScript 编写，也就是说，假如你不理解或者你想了解更多细节，你可以直接阅读 Node 的源码。这不但包括像网络、文件操作、模块系统，以及 stream 这些模块，还包括 Node 特有的特性，例如，通过 cluster 模块同时运行多个 Node 进程，以及可以将代码片段封装在事件驱动的异常处理中的 domain 模块。

接下来，我们将从事件开始深入每个核心模块。

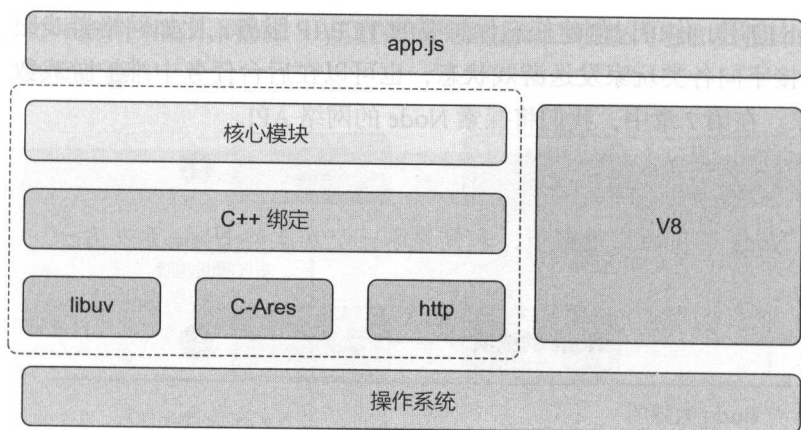


图 1.2 Node 环境中的关键部分

EventEmitter 事件的接口

每个 Node 开发者迟早会碰到 `EventEmitter`，一开始，它像是那些只有类库开发者才会使用的东西，但实际上它是大多数 Node 核心模块的基础，`Stream`、网络、文件系统全部继承于它。

你可以基于 `EventEmitter` 来创建自己基于事件 API，例如你要开发一个 `paypal` 付款处理的模块，你可以让它基于事件，这样 `Payment` 对象的实例可以触发像 `paid` 和 `refund` 这样的事件，通过这样的设计，你可以将这个模块从你的业务逻辑中分离出来，让它可以在其他项目被重用。

我们在第 4 章有一整章的内容来讲解事件。另外一个有意思的地方是，`stream` 模块也是基于 `EventEmitter` 的。

Stream：高可扩展性 I/O 的基础

`Streams` 继承于 `EventEmitters`，能被用来在不可预测的输入下创建数据，比如网络连接，数据传输速度取决于其他用户正在干什么。通过 Node 的 `stream` API，你可以创建一个对象接收关于连接的事件，在接收到新数据时触发 `data` 事件，在结束连接时触发 `end` 事件，在有错误发生时触发 `error` 事件。

相比较把许多的回调函数传入一个 `readable stream` 的构造函数，你只订阅你关心的事件要好得多，多个 `streams` 也可以连接起来，这样你可以用一个 `stream` 对象从网络读取数据，把读取到的数据输送到另外一个 `stream` 中加工成另外一个对象，可以把 `xml` 文件的数据读取出来转换成 `JSON` 格式，让 JavaScript 操作起来更容易。

我们喜欢 Stream，所以，为它准备了一整个章节。你可以直接跳到第 5 章去深入了解它。你可能觉得 stream 和事件听上去很抽象，没错，它们的确很抽象，但它们是 I/O 模块（例如文件系统和网络）的基础。

FS：处理文件

Node 的文件模块不但可以通过非阻塞的 I/O 读写文件，而且它也有同步的方法。你可以通过 `fs.stat` 异步获取文件的信息，也可以通过 `fs.statSync` 同步读取。

如果你想通过 Stream 的方式高效地处理文件内容，那么你可以通过 `fs.createReadStream` 来获得一个 `ReadableStream` 对象。在第 6 章，我们会更多地讨论这个方法。

网络：创建网络客户端与服务端

网络模块是 http 模块的基础，也可以用来创建通用的网络客户端与服务端。尽管 Node 开发通常指的是 web 开发，在第 7 章你会看到如何创建 TCP 和 UDP 的服务，这意味着你并不局限于 http 开发。

全局对象与其他模块

假如你有用 Node 开发 web 应用的经验，也许是 Express 框架，那么你也许并不知道你已经使用了 `http`、`net` 以及 `fs` 等核心模块。其他的内置模块也许不那么吸引眼球，但也是至关重要的。

全局对象与方法的设计就是其中一例，比如 `process` 对象，它让你可以把数据传入或者传出标准 I/O 流（`stdout`、`stdin`）。就像在 UNIX 或者 Windows 脚本中，你可以把数据通过 `cat` 直接传给 Node 程序。还有无处不见的 `console` 对象，所有的 JavaScript 开发都爱它，也是一个全局对象。

Node 的模块系统也是全局方法。在第 2 章，你将看到如何使用这些特性。

现在你已经了解了一些核心模块，是时候看看怎么使用它们了。下面的例子，我们将会使用 `stream` 模块来生成文件流的数据，你可以用在文件和 http 连接上。如果你想了解更多关于 `stream` 和 `http` 的基础，那请参阅 *Node.js in Action*。

1.2 构建一个 Node 应用

不多废话，我们将带着你一起构建一个 Node 应用。虽然它不是什么大应用，但它会用到一些 node 的核心功能，比如模块和流，这将是一个快速紧张的过程，那么，打开你的编辑器和终端，开始吧。

接下来的 10 分钟你将学到：

- 如何开始一个新的 Node 项目
- 如何创建你自己的 Stream 类
- 如何编写运行一个简单的测试

Stream 非常擅长处理数据，无论是读、写或者是转换。想象一下，你需要把数据从数据库导入到另外一个格式，比如 csv。你可以创建一个 Stream 类，接收从数据库输入的数据，并将它输出到 csv 的流中，这个新的 csv 流可以传入一个 http 请求，这样，这个 csv 的输出可以直接显示在浏览器上。它还可以接入一个可写的文件流，你甚至可以用这个流来创建一个文件，并发送至浏览器。

在这个例子中，这个 Stream 类可以接受文本输入，记录通过正则表达式匹配的单词，在文件流发送完后把结果通过一个事件发布出去。你可以用它来记录文件中匹配的单词，也可以把网页内容传入这个流，来记录 p 标签的数量，这都取决于你，但首先你需要创建一个新的项目。

1.2.1 创建一个新的 Node 项目

你也许在猜专业的 Node 开发如何创建一个新的项目，因为有 npm 在，这是个非常简单的过程。虽然你可以创建一个 JavaScript 文件，并且执行 `node file.js`，我们将使用 `npm init` 来创建一个带有 `package.json` 文件的新的项目。创建一个新的目录^❶，`cd`^❷进去，然后运行 `npm init`^❸。

```
mkdir first-project
cd first-project
npm init
```

❶
❷
❸

- ❶ 创建一个新的目录。
- ❷ 进入目录中。
- ❸ 创建项目的清单文件。

要习惯敲这些命令，你会经常使用它们！你可以按回车键来接受 npm 提供的默认值。在写 JavaScript 代码之前，你已经领略了 Node 最主要的功能之一——npm 是多么酷。它不但可以安装模块，也可以用来管理项目。

现在是时候写 JavaScript 了。在下一节，你将创建一个新的 JavaScript 文件来实现一个 Stream 类。

何时使用 package.json 文件

你可能想写一个小脚本，在犹豫是否需要创建一个 package.json 文件。它并不是必需的，但通常你应该尽可能地创建它们。

Node 开发者倾向于小模块，在 package.json 中写明依赖意味着你的项目无论多小，在未来或者其他开发者的机器上是非常容易安装的。

1.2.2 创建一个流的类

创建一个名为 countstream.js 的新文件，使用 util.inherits 来继承 stream.Writable 并且实现必要的 _write 方法。太快了点？我们缓一缓，先看看下边的源码。

例子 1.1 一个用于计数的可写流

```
var Writable = require('stream').Writable;
var util = require('util');
```

```
module.exports = CountStream;
```

```
util.inherits(CountStream, Writable);
```

```
function CountStream(matchText, options) {
  Writable.call(this, options);
  this.count = 0;
  this.matcher = new RegExp(matchText, 'ig');
}
```

```
CountStream.prototype._write = function(chunk, encoding, cb) {
  var matches = chunk.toString().match(this.matcher);
  if (matches) {
    this.count += matches.length;
  }
  cb();
};
```

```
CountStream.prototype.end = function() {
  this.emit('total', this.count);
};
```

❶ 继承可写流。

❷ 创建一个全局且忽略大小写的正则对象。

❸ 把当前的输入数据转化为字符串并进行匹配。

❹ 当输入流结束时，触发 `total` 事件。

这个例子简单地展示了书中后续的例子是怎么样的。我们给出了一个代码片段，并且附加了代码的注释。例如，第一部分代码的类使用了 `util.inherits` 方法来继承一个 `Writable` 基类❶。这个例子没有完整地在这里展示，更多的可以参考第 5 章的第 30 个技巧。现在，我们把重点放在正则表达式是如何传递给构造函数❷以及如何文本流入类的示例时对其计数❸的。Node 的 `Writable` 类为我们调用 `_write` 方法，暂时可以不用去关注它。

Streams 和事件

在例子 1.1 中有一个事件，`total`。这是我们自己创建的——当然你也可以自己创建一个。`Streams` 从 `EventEmitter` 继承而来，所以它们都拥有同样的 `emit` 和 `on` 方法。

当不再有数据时，Node 的 `Writable` 基类会调用 `end` 方法❹。该类可以被实例化，并且根据需要来通过管道进行传输数据。下一节你会了解到如何使用管道来连接多个流。

1.2.3 使用流

现在你已经知道如何创建一个流的类，很可能你已经实践过了。现在创建一个新的文件，`index.js`，并加入如下的代码。

例子 1.2 使用 `CountStream` 类

```
var CountStream = require('./countstream');  
var countStream = new CountStream('book');  
var http = require('http');  
  
http.get('http://www.manning.com', function(res) {  
  res.pipe(countStream);  
});  
  
countStream.on('total', function(count) {  
  console.log('Total matches:', count);  
});
```

❶ 加载 `countstream.js`。

❷ 创建一个 `CountStream` 的示例用于匹配 `book` 的文本计数。

❸ 下载 www.manning.com 页面。

❹ 从网站中以管道的方式把数据传给 `countStream` 用于文本计数。

你可以尝试着使用 `node index.js` 来执行这个例子的代码。它应该会打印 `Total matches: 24` 之类的。你可以尝试着更改一下所要抓取的地址。

这个例子加载了例子 1.1 ❶ 中的模块并通过字符串 `'book'` 将它实例化❷。同时使用了 Node 的标准模块 `http` 从网站下载文本内容❸以及使用管道把结果传入到我们的 `CountStream` 类中❹。

这里最重要的是 `res.pipe(countStream)`。当你使用管道传输数据时，不用去关心数据有多大或者网络速度多慢：`CountStream` 类会完整进行匹配计数直到数据全被处理完。这个 Node 的程序并不会在一开始下载整个文件！它会把文件一块一块地进行处理。这是很重要的，也是 Node 提供的一个关键的特性。

总结一下，图 1.3 概括了到现在为止，创建一个新的 Node 程序需要做什么。首先创建一个新的目录，执行 `npm init` 命令❶，然后创建一些 JavaScript 代码文件❷，最后执行代码❸。

❶ 创建一个新的目录然后运行
`npm init`

```
$ mkdir new-project
$ cd new-project
$ npm init
```

`index.js`

❷ 创建 JavaScript 文件

❸ 运行代码

```
$ node index.js
$ npm start
```

图 1.3 创建新的 Node 项目的 3 个步骤

关于 Node 环境，另外一个很重要的便是测试，下一章将会给出测试 `CountStream` 的例子。

1.2.4 编写测试

我们可以不借助任何第三模块来为 `CountStream` 编写一个简单的测试。Node 已经内置了 `assert` 模块用于提供断言，我们可以用它来进行快速的测试。创建 `test.js` 并且加入如下代码。

例子 1.3 使用 CountStream 类

```
var assert = require('assert');
var CountStream = require('./countstream');
var countStream = new CountStream('example');
var fs = require('fs');
var passed = 0;
```

```
countStream.on('total', function(count) {
  assert.equal(count, 1);
  passed++;
});
```

```
fs.createReadStream(__filename).pipe(countStream);
```

```
process.on('exit', function() {
  console.log('Assertions passed:', passed);
});
```

❶ 当流结束时，*total* 事件将会被触发。

❷ 断言所预计的计数。

❸ 为当前文件创建一个可读流，并且把数据通过管道传给 *CountStream*。

❹ 在程序结束前，展示执行了的断言的数量。

这个测试代码可以使用 `node test.js` 来执行，并且你会看到在控制台打印出 `Assertions passed: 1`。这个测试代码读取当前文件，并且将其内容传给 *CountStream*。这个例子有点问题，但是在这个场景下，它卓有成效，我们可以一下子就确定程序会匹配到一个单词 *example*。

断言

Node 内置了一个叫 *assert* 的断言类。一个最简单的测试可以直接使用这个类来构建——`assert(表达式)`。

测试的例子首先做的是监听 *CountStream* 的实例会触发的 *total* 事件❶。这里可以很好地对所期望的匹配数进行断言❷。一个可读流将把关联的当前打开的文件的数据以管道的方式传给我们的类❸。然后，在程序结束前，我们可以打印出多少断言是正确的❹。

有一点非常重要，如果 *total* 事件永远不会被触发，那么 `assert.equal` 也就不会执行。我们便没有办法知道在回调函数中的测试是否执行了，所以这里使用了很常见的计数器来

进行标示。在 Node 程序中也同样可以使用你可能已经很熟悉的其他语言或者平台中常见的编程模式。

如果你有点厌倦了，那么可以休息下。但还有一些好东西来帮助我们更好地实现项目。Node 开发者喜欢在命令行里使用 `npm` 命令来跑测试或者其他脚本。打开 `package.json`，把 `"test"` 属性改成这样子：

```
"scripts": {  
  "test": "node test.js"  
},
```

现在就可以输入 `npm test` 来运行我们的测试了。当你有很多测试需要运行或者运行测试变得复杂时，这可以帮助你更加方便地来处理测试。运行测试，测试运行环境，异步测试等相关内容都会在第 10 章讲述。

npm scripts

`npm test` 和 `npm start` 命令可以在 `package.json` 中进行配置。你也可以使用 `npm run command` 来运行自定义的一些命令。这需要你像上述例子一样在 `scripts` 属性中进行配置。

这在针对特定类型的测试或者经常要反复执行的程序时特别有用——如 `npm run integration-test`，执行集成测试，甚至是 `npm run seed-data` 来生成种子数据。

如果你先前缺少关于 Node 开发的经验，那么这个例子可能有点难，但是它让我们了解 Node 开发者是如何思考问题的，同时，它向我们展示了 Node 强大的优势。

现在，通过以上的讲解，我们可以认识到一个 Node 项目是如何被构建起来的。下一章中将会介绍我们一开始会使用到的技术点，会涉及到在所有 Node 程序中使用到的全局特性。

1.3 总结

在这一章中，你已经了解了本书涵盖的内容，以及它如何聚焦 Node 中一些强大的内置核心模块，如网络模块、文件系统模块。

你也了解到 Node 为什么那么火，以及如何使用 Node。我们已经提及的有以下多个方面：

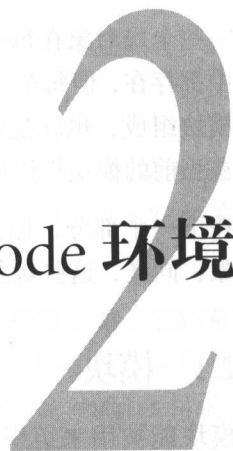
- 什么时候适合使用 Node，以及 Node 如何在非阻塞 I/O 的基础上来构建一个高性能的标准 JavaScript 的执行环境。

- Node 的标准库，也称为它的核心模块。
- 核心模块如何处理像网络协议、文件读写或者像流这样更通用的特性中的 I/O。
- 快速开始创建一个 Node 项目以及利用 `package.json` 来添加依赖和可执行脚本。
- 使用 Node 强大的 `stream` API 来处理数据。
- `Streams` 类继承于 `EventEmitter`，你可以使用它来触发或者响应你需要使用的各种事件。
- 利用 `npm` 和 `assert` 模块来编写一个简单的测试，而无须依赖任意第三方模块。

总的来说，我们希望你已经从简单的入门应用学习到一些东西，虽然 Node 无非就是使用基于事件的 API、非阻塞的 I/O，以及 `stream`，但是利用 Node 提供的独特的工具，像 `package.json` 配置文件以及 `npm` 也很重要。

又回到讨论技术的时间了，下一章会介绍不用额外加载就可以使用的全局对象。

全局变量：Node 环境



本章概要

- 使用模块
- 不使用其他模块你能做什么
- process 与 console 对象
- Timers

全局对象能被用在所有的模块中，无论你是写一个网络应用、命令行脚本还是网站应用，你的程序都能访问这些对象。这意味着，你永远可以依赖像 `console.log` 与 `__dirname` 这些特性，我们会在本章详细解释这两个特性。

这一章节的目的是介绍 Node 的全局对象与方法，来帮助你学习那些能被所有 Node 进程所使用的方法。这能帮助你更好地理解 Node、它与操作系统的关系，以及它和其他 JavaScript 运行时的区别，比如浏览器。

Node 提供一些重要的内置方法，甚至不用加载其他的模块。除了那些 ECMAScript 提供的功能，Node 有几个由 Node 提供的 host 对象，来帮助 Node 程序执行。

一个重要的全局对象是 `process`，它可以用来与操作系统通信。UNIX 开发者所熟悉的标准输入输出流，可以通过 Node 的 `streaming` 接口从 `process` 对象获取。

另一个重要的全局对象是 `Buffer` 类。它的出现是由于 JavaScript 天生缺乏对二进制数据的支持。这个问题已经在 ECMAScript 标准的发展下得到了解决,但现在大多数 Node 开发者依赖 `Buffer` 类。我们会在第 3 章详细介绍 `Buffer` 类。

一些全局对象在每一个模块中都是独立的实体。比如说 `module` 对象在所有的 Node 程序中都存在,但每个 `module` 对象仅仅存在于它当前的模块中。因为 Node 程序可以由几个模块组成,也就是说一个程序有几个不同的 `module` 对象,它们类似 `globals` 对象,但只在当前的模块作用域下。

在下一个部分,你会学习到如何加载模块。那些与模块相关的方法与对象都是全局对象,同样,这些对象能被直接拿来就用。

2.1 模块

模块能被用来组织大型的项目以及分发 Node 项目,所以了解如何安装与创建它非常重要。

技巧 1 安装与加载模块

无论你是使用 Node 提供的核心模块还是从 npm 安装的第三方模块,对模块的支持是整合在 Node 中的,是一直可用的。

问题

你想从 npm 加载一个第三方模块。

解决方案

从命令行工具 npm 安装模块,然后通过 `require` 加载模块。下面的例子展示了如何安装 `express` 模块。

例子 2.1 使用 npm

```
$ npm search express
express          Sinatra inspired web development framework
$ npm install express
express@x.x.x ./node_modules/express
├── methods@x.x.x
└── (Several more dependencies appear here)

$ node
> var express = require('express');
```

```
> typeof express  
'function'
```

- ❶ 通过关键字查找模块。
- ❷ 通过 `require` 方法加载模块。

讨论

`npm` 命令行工具是和 `Node` 一起发布的，可以被用来查找、安装与管理包。网站 <https://npmjs.org> 提供了另外一个界面来查找模块，每个模块都有它自己的页面来展示它相关的 `readme` 文件以及它的依赖文件。

一旦你知道了模块的名字，安装就变得简单：通过输入 `npm install 模块名` ❶，它将被安装在 `./node_modules` 目录下，模块也能被全局安装，运行 `npm install -g 模块名` 将把模块安装到全局的目录下。在 `UNIX` 系统中它通常是 `/usr/local/lib/node_modules` 目录。在 `Windows` 系统中，它与 `node.exe` 文件同一目录。

在模块被安装之后，可以通过 `require('模块名')` 来加载模块❷。`require` 方法通常返回一个对象或者一个方法，这取决于模块是如何创建的。

搜索 npm

默认情况下，`npm` 通过模块的 `package.json` 文件中的几个字段进行查询。这包括模块名、描述、开发者、`Url`，以及关键字。一个简单的查询，例如 `npm search express`，将返回成百上千的数据。

你可以通过正则表达式来减少查询结果。将查询条件包括在两个斜杠中来通过正则表达式查询：`npm search /^express$/`。

然而，这也不能解决所有问题，幸好有开源的模块来增强它内置的查询命令。比如 `Gorgi Kosev` 开发的 `npmsearch` 可以通过它的相关度来进行排序。

是否将模块安装成全局模块对开发可维护的项目至关重要。假如有其他开发者需要和你一起开发，那么你应该考虑将模块作为依赖添加到你项目中的 `package.json` 文件。严格地管理项目的依赖可以在将来当依赖模块有新版本发布时，管理它们变得容易许多。

技巧2 创建与管理模块

除了安装与分发开源项目，本地模块可以用来组织项目。

问题

你想将一个项目解耦至多个文件。

解决方案

通过 `exports` 对象。

讨论

Node 的模块系统提供了将代码拆分成多个文件的解决方案。这与 C 语言中的 `include` 有着很大的不同, 甚至与 Ruby 和 Python 中的 `require` 也不同。主要的不同是, Node 中的 `require`, 返回了一个对象而不是像 C 语言中的预处理器那样把代码加载到当前的命名空间中。

在技巧 1 中, 你看到如何通过 `npm` 来安装模块, 以及 `require` 是如何被用来加载它们。要管理模块不仅仅是 `npm` 的工作, Node 有一套内置的模块系统, 基于 CommonJS `modules/1.1` 标准 (<http://wiki.commonjs.org/wiki/Modules/1.1>)。

对象、方法以及变量都可以从一个文件 `export` 出来用在其他任何地方。`exports` 对象一直存在, 虽然这一章节主要在探索全局对象, 但它并不是真正意义上的全局对象, 准确地说 `exports` 对象的作用域是在一个模块中。

当一个模块只存在一个类时, 那么使用模块的开发者喜欢 `var MyClass = require('my-class');` 而不是 `var MyClass = require('myclass').MyClass`, 所以你应该使用 `modules.export`。例子 2.2 展示了它如何工作。这和使用像 `exports` 那样需要你设置一个属性来导出模块不同。

例子 2.2 导出模块

```
function MyClass() {  
}
```

```
MyClass.prototype = {  
  method: function() {  
    return 'Hello';  
  }  
};
```

```
var myClass = new MyClass();
```

```
module.exports = myClass;
```

❶

❶ 对象可以和其他对象、方法以及属性一起被 `export` 出来。

例子 2.3 展示了如何导出多个对象、方法或者变量，这个技巧通常被用作导出多个对象的工具类。

例子 2.3 对外导出多个对象、方法和值

```
exports.method = function() {  
    return 'Hello';  
};  
  
exports.method2 = function() {  
    return 'Hello again';  
};
```

最后，例子 2.4 展示了如何通过 `require` 来加载这些模块，以及如何使用它们所提供的方法。

例子 2.4 使用 `require` 加载模块

```
var myClass = require('./myclass');  
var module2 = require('./module-2');  
  
console.log(myClass.method());  
console.log(module2.method());  
console.log(module2.method2());
```

❶

❷

❶ 加载 `myclass.js`。

❷ 加载 `module-2.js`。

要注意的是加载一个本地模块需要加上路径名，在这些例子中，这个路径是 `./`。没有它的话，Node 将试图在 `$NODE_PATH` 中寻找符合的模块，然后是 `./node_modules`、`$HOME/.node_modules`、`$HOME/.node_libraries`，或者 `$PREFIX/lib/node`。

在例子 2.4 中需要注意的是，`./myclass` 自动地加上了后缀 `./myclass.js`❶，`./module-2` 展开成 `./module-2.js`❷。

这个程序的输出是：

```
Hello  
Hello  
Hello again
```

哪个模块?

要判断 node 具体加载了哪个模块, 可以通过 `require.resolve(id)`。这将返回文件的绝对路径。

一旦一个模块被加载了, 它将被缓存, 也就是说, 多次加载它将会返回同一个对象。这通常是高效的, 这会帮助你在一个项目中重用模块而不用担心使用 `require` 时的开销。你可以安全地在同一模块中调用 `require` 而不用集中加载所有的依赖。

卸载模块

虽然自动缓存在 Node 开发中适合大多数场景, 也存在着一些情况你希望卸载一个模块, 可以通过 `require.cache` 对象来达到这个目的。

要从缓存中删除一个模块, 可以使用 `delete` 关键字。还需要模块的绝对路径, 可以通过 `require.resolve` 来获取。例如:

```
delete require.cache(require.resolve('./myclass'));
```

这将返回 `true`, 代表模块被卸载了。

在下一个技巧中, 你将学到如何聚合相关的模块并且一次性加载它们。

技巧3 加载一组相关的模块

Node 可以将目录作为模块, 可以把相关模块按相关逻辑组合起来。

问题

你希望将一个目录下的相关文件组合起来, 而且只需要通过一个 `require` 来加载这些模块。

解决方案

创建一个叫作 `index.js` 的文件来加载各个模块并把它们一起导出, 或者在文件夹下添加一个 `package.json` 文件。

讨论

通常一个模块是逻辑自包含的, 但将它拆分成几个文件还是很有意义的。大多数你在 npm 上找到的模块是以这种方式编写的。Node 的模块系统可以将文件目录作为模块。最容易的是创建一个叫 `index.js` 的文件, 这个文件通过 `require` 来加载各个文件。下面的例子展示了这是如何做到的。

例子 2.5 group/index.js 文件

```
module.exports = {  
  one: require('./one'),  
  two: require('./two')  
};
```

❶ 这个模块将当前目录下的各个文件组合起来一起导出。

group/one.js 以及 group/two.js 可以导出方法或者变量❶。下一例子展示了这样一个文件。

例子 2.6 group/one.js 文件

```
module.exports = function() {  
  console.log('one');  
};
```

需要把目录作为模块使用的代码，通过一个 require 语句来一次性加载所有的模块。下面的例子展示了如何加载目录模块。

例子 2.7 加载一组模块

```
var group = require('./group');
```

```
group.one();  
group.two();
```

❶ 这个 require 语句不需要特殊的处理来加载一组模块。

例子 2.7 的输出应该是这样的：

```
one  
two
```

这个方法通常被用来作为组织 web 应用的架构技巧。相关的对象，比如控制器、模型以及视图，都可以放在不同的目录下来拆分应用。图 2.1 展示了如何通过这个模式来组织应用。

Node 还提供了一个替代方案来实现这个模式。在目录下添加一个 package.json 文件可以帮助模块系统解析出如何一次性加载目录下所有的文件。这个 JSON 文件应该包含一个 main 属性来指向一个 JavaScript 文件。实际上这是 Node 在加载模块时找的默认的文件，如果 package.json 不存在，那么它就会去找 index.js。下面展示了 package.json 的一个例子。

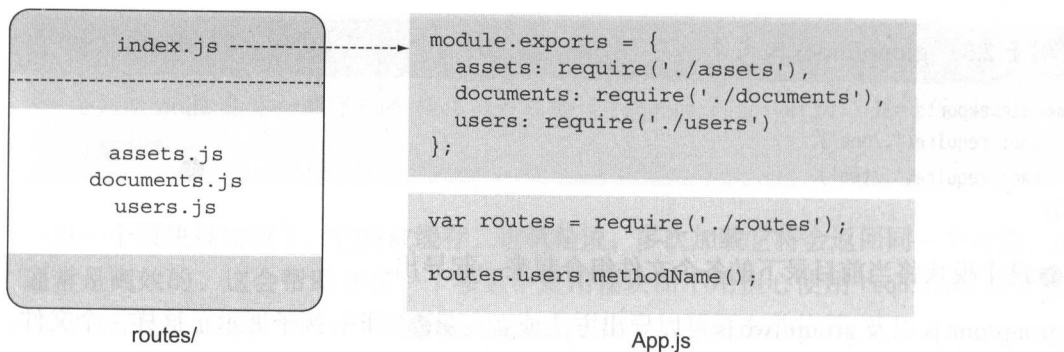


图 2.1 目录作为模块使用

例子 2.8 包含了一个模块的目录中的 package.json 文件

```
{ "name" : "group",
  "main" : "../index.js" }
```

①

① 这个可以指向任意文件。

文件后缀

当加载一个文件时, Node 会找以 .js、.json, 以及 .node 为后缀的文件。require.extensions 数组可以用来通知 require 去加载其他类型的文件。Node 的模块系统在将目录作为模块时也会把这个算进去。

这个特性在 Node 文档中已经被表示为弃用, 但是模块系统也被标记成 “lock”, 所以它应该还能用。如果你想使用这一特性, 你应该先查看 Node 的文档。^a如果你仅尝试从老系统中加载一个有特殊后缀的 JavaScript 文件, 那么这个特性非常适合尝试一下。

^a参见 http://nodejs.org/api/globals.html#globals_require_extensions

require API 提供了很多管理文件的方法。但当你想要加载一些相对于当前模块或者目录的模块时, 该怎么办? 继续读下去, 答案将在技巧 4 中揭晓。

技巧 4 使用路径

有些时候, 你需要根据相对的路径来打开文件。Node 提供了类库来找到当前路径、目录或者模块的路径。

问题

你希望打开一个不在模块系统中的文件。

解决方案

通过 `__dirname` 或者 `__filename` 来找到文件的位置。

讨论

有些时候，你需要从一个不在模块系统中的文件加载数据，但你需要通过当前脚本的路径来找到那个文件，比方说，web 应用中的模板文件。`__dirname` 和 `__filename` 变量在这种场景下非常有用。

运行下面的例子可以输出这些值。

例子 2.9 路径变量

```
console.log('__dirname:', __dirname);  
console.log('__filename:', __filename);
```

❶

❶ 这些变量保存着当前脚本的绝对路径。

大多数的开发者通过拼接字符串的方法来组合这些变量与路径片段：`var view = __dirname + '/views/view.html'`。这在 Windows 和 UNIX 的操作系统上都能行得通，Windows API 足够聪明，以至于它能将斜杠转换为它的标准格式，所以你不需要特殊的处理来支持这两个操作系统。

假如你真的希望确保路径是拼接正确的，你可以使用 `path` 模块的 `path.join` 方法：`path.join(__dirname, 'views', 'view.html')`；。

除了模块管理，还有用来写入标准输入流全局对象。下一组技巧将探索 `process.stdout` 以及 `console` 对象。

2.2 标准 I/O 以及 console 对象

在 UNIX 或者 Windows 系统中，文字可以通过命令行工具导入 Node 进程。这个章节包括了如何使用这些标准 I/O 流的技巧，以及如何正确地使用 `console` 对象来完成日志相关的工作。

技巧5 标准 I/O 流的读写

无论你需要从一个应用中读取数据, 还是写入数据, 通过 `process` 对象来读写 I/O 流是一个有用的技巧。

问题

你希望从一个 Node 程序导出或导入数据。

解决方案

使用 `process.stdout` 与 `process.stdin`。

讨论

`process.stdout` 对象是一个可写的 `stream`。我们会在第 5 章详细地讲解 `stream`, 现在你只需要知道, 它是 `process` 的一部分, 所有 Node 程序都可以使用它, 并且在显示与接受输入时非常有用。

下面的例子展示了如何从命令行将文字接入 `node` 程序, 处理它, 并把它再一次输出。

例子 2.10 Path 变量

```
// Run with:  
// cat file.txt | node process.js  
  
process.stdin.resume();  
process.stdin.setEncoding('utf8');  
  
process.stdin.on('data', function(text) {  
  process.stdout.write(text.toUpperCase());  
});
```

❶

❷

❶ 通知 `stream` 准备开始读取数据。

❷ 这个回调以块的形式处理数据。

每次一个数据库从输入流读入时, 它会被 `toUpperCase()` 转换, 然后写入输出流。图 2.2 展示了数据如何从操作系统的进程中通过你的 Node 程序, 然后输出到另一个程序中的过程。在终端中, 这些程序可以通过管道符号 `|` 连接在一起。

这种基于管道的方法在 UNIX 操作系统中处理输入非常好用, 因为其他许多命令也是这样设计的。这给 Node 应用带来了像乐高积木那样的模块化, 使得重用变得很容易。

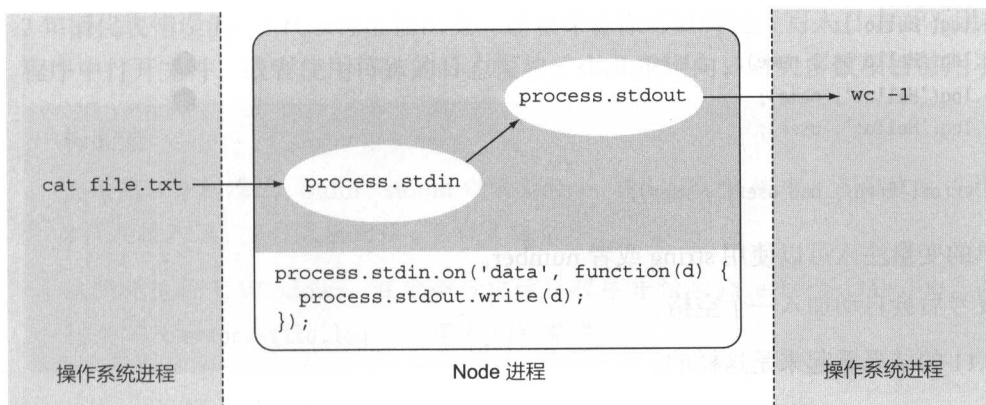


图 2.2 使用标准输入输出的一个简单程序的数据流

假如你仅仅希望打印消息或者错误,那么 Node 提供了一个更加简单的接口,通过 `console` 对象来为这个目的量身定做。下一个技巧说明了如何使用它,以及它的隐藏特性。

技巧 6 打印日志消息

在一个程序中打印日志或者错误信息最简单的方法是通过 `console` 对象。

问题

你希望记录不同类型的消息到 `console`。

解决方案

使用 `console.log`、`console.info`、`console.error`, 以及 `console.warn`, 一定要利用这些方法内置的格式工具。

讨论

`console` 对象有几个方法能被用来输出不同类型的消息。它们会被写入相关的输出流中,这意味着你可以在 UNIX 系统中相应地 `pipe` 它们。

虽然它基本上就一行代码 `console.log('message')`, 但其中还有更多的功能。变量能被填入 `message` 中, 或简单地附加字面量, 这使得记录消息来显示变量或者对象的内容变得异常简单。下面的例子展示这些特性。

例子 2.11 Path 变量

```
var name = 'alex';
var user = { name: 'alex' };
```

```
console.log('Hello');
console.log('Hello %s', name);
console.log('Hello:', name);
console.log('Hello:', user);

console.error('Error, bad user:', user);
```

- 1
- 2

- 1 简单的变量注入可以使用 string 或者 number。
- 2 在冒号后会自动加入一个空格。

例子 2.11 的结果看起来是这样的:

```
Hello
Hello alex
Hello: alex
Hello: { name: "alex" } //
Error, bad user: { name: 'alex' }
```

- 1

- 1 user 对象被 util.inspect 格式化。

消息内容的格式化是通过 util.format 实现的。表 2.1 显示了它支持的格式。

表 2.1 格式占位符

占位符	类型	例子
%s	String	'%s', 'value'
%d	Number	'%f', 3.14
%j	JSON	'%j', { name: 'alex' }

这些格式占位符非常方便, 简单地将对象引入 console.log 中而不用手动拼接字符串, 在记录消息时特别好用。

info 和 warn 方法是 log 与 error 的同义词。log 与 error 的区别是输出流的不同。在技巧 5 中, 你看到 Node 如何在所有程序中使用标准输入输出流。它还通过 process.stderr 暴露了错误流。console.error 方法将写入这个流而不是 process.stdout。这意味着你可以在终端或者 shell 脚本中将 Node 程序的错误消息重定向。

假如你通过 2> error-file.log 执行前一个例子, 错误消息将会被重定向到 error-file.log 文件。其他的消息还是会按原来那样打印在终端中。

```
node listings/globals/console-1.js 2> errors-file.log
```

2 句柄代表错误流；1 代表标准输出流。这意味着你可以将错误写入日志而不用在 Node 程序中打开文件，或者使用特殊的日志模块。shell 重定向，对大多数项目够用了。

标准流

总共有 3 个标准流，`stdin`、`stdout` 以及 `stderr`。在 UNIX 终端中，它们被数字代表。0 代表输入流，1 代表输出流，2 代表错误。

这同样适用于 Windows：在命令行中运行程序并加入 `2> errors-file.log`，将把错误写入 `errors-file.log`，就像 UNIX 那样。

堆栈追踪

另一个 console 对象的特性是 `console.trace()`。这个方法在当前执行点产生堆栈日志。产生的堆栈轨迹包含了执行异步回调的代码的行号，这将帮助定位异常，不然，这将很难追踪。比如在事件监听器中产生的堆栈轨迹将会显示这个事件是在哪里被触发的。第 5 章的技巧 28 将会详细地探索它。

另一个比较有用的是 `benchmarking` 的功能。接下来深入了解。

技巧 7 基准测试

Node 可以在不依赖其他工具的情况下完成基准测试。

问题

你需要对一个耗时的操作进行基准测试。

解决方案

使用 `console.time()` 和 `console.timeEnd()`。

讨论

作为一个 Node 开发者，有时你需要判断为什么一个操作会很慢。幸运的是，console 对象自带一些内置的基准测量特性。

调用 `console.time('label')`，以毫秒记录当前的时间，并且在后来调用 `console.timeEnd('label')` 显示从那一点开始的持续时间。整个持续的时间会自动地以毫秒和那个标签打印在一起，所以你不需要额外地调用 `console.log` 来打印标签。

例子 2.12 是一个简短的程序，它接受命令行参数（参考技巧 9 来了解更多如何处理参数），通过基准测试来看读取文件有多快。

例子 2.12 基准测试一个函数

```
var args = {
  '-h': displayHelp,
  '-r': readFile
};

function displayHelp() {
  console.log('Argument processor:', args);
}

function readFile(file) {
  if (file && file.length) {
    console.log('Reading:', file);
    console.time('read');
    var stream = require('fs').createReadStream(file)
    stream.on('end', function() {
      console.timeEnd('read');
    });
    stream.pipe(process.stdout);
  } else {
    console.error('A file must be provided with the -r option');
    process.exit(1);
  }
}

if (process.argv.length > 0) {
  process.argv.forEach(function(arg, index) {
    args[arg].apply(this, process.argv.slice(index + 1));
  });
}
```

❶ 调用 `console.timeEnd()` 会显示基准测试信息。

通过多次交叉调用 `console.time` 与不同的标签，可以产生多次基准测试，这对探究复杂、多层嵌套的异步函数的性能是非常有用的。

这些方法基于 `Date.now()` 计算函数执行时间，精确到毫秒。要获取更加准确的基准，可以使用第三方模块 `benchmark` (<https://npmjs.org/package/benchmark>)，并结合 `microtime` 模块 (<https://npmjs.org/package/microtime>)。

`process` 对象能被用来操作标准 I/O 流。如被正确地使用，`console` 对象能处理许多那些缺乏经验的开发者使用第三方模块处理的任务。在下一节，我们将会探索 `process` 对象，来看它是如何与操作系统集成的。

2.3 操作系统与命令行

`process` 对象能被用来获取操作系统的信息，并且通过退出码（`exit code`）、信号量（`signal`）与其他进程进行通信。这一节包含了一些如何使用它们的高级技巧。

技巧 8 获取平台信息

Node 有一些内置的方法来查询操作系统。

问题

你需要基于操作系统或者处理器架构运行特定于平台的代码。

解决方案

使用 `process.arch` 与 `process.platform` 属性。

讨论

Node 通常是跨平台的，所以你一般不需要处理跨平台或者处理器。但你希望利用系统的特性来调整项目，或者简单地收集这个脚本跑在什么系统上的数据。某些基于 Windows 的模块可以在 32 位或 64 位的二进制文件间切换。下面的例子展示了这是如何做到的。

例子 2.13 基于系统架构的分支

```
switch (process.arch) {  
  case 'x64':  
    require('./lib.x64.node');  
    break;  
  case 'ia32':  
    require('./lib.Win32.node');  
    break;  
  default:  
    throw new Error('Unsupported process.arch:', process.arch);  
}
```

另外一些来自系统的信息可以通过 `process` 模块搜集。其中一个方法是 `process.memoryUsage()`——它返回一个有 3 个属性的对象来描述当前进程的内存使用情况。

- `rss`——常驻内存大小，是指在 RAM 中维持的进程内存的那一部分
- `heapTotal`——动态分配的可用内存
- `heapUsed`——已经使用的堆大小

下面一个技巧详细地展示了如何处理命令行参数。

技巧9 传递命令行参数

Node 提供了一个简单的 API 来处理命令行参数，你可以用来传入程序。

问题

你正在写一个需要从命令行接收简单参数的程序。

解决方案

使用 `process.argv`。

讨论

数组 `process.argv` 允许你检查是否有参数传入你的脚本。因为它是一个数组，你可以用它来查看有多少参数被传入，如果有，那么头两个参数是 `node` 以及这个脚本的名字。

例子 2.14 展示了使用 `process.argv` 的一个方法。这个例子循环遍历 `process.argv`，并将它们解析且传入方法中，你可以执行脚本 `node arguments.js -r arguments.js`，这将会把这个文件的代码打印出来。

例子 2.14 操作命令行参数

```
var args = {  
  '-h': displayHelp,  
  '-r': readFile  
};  
  
function displayHelp() {  
  console.log('Argument processor:', args);  
}  
  
function readFile(file) {  
  console.log('Reading:', file);  
  require('fs').createReadStream(file).pipe(process.stdout);  
}  
  
if (process.argv.length > 0) {  
  process.argv.forEach(function(arg, index) {  
    args[arg].apply(this, process.argv.slice(index + 1));  
  });  
}
```

❶ 这是一个简单的对象，用来表示可用的参数。

❷ 使用管道把文件发送到标准输出流。

③ 调用从参数模型中匹配到的方法，并通过对完整的参数列表进行分割来有效支持来自命令行标记的选项传递。

`args` 对象①保存了脚本支持的开关。然后 `createReadStream` ②用来将文件传入标准输出流。最后，命令行参数所对应的函数通过 `Function.prototype.apply` 执行③。

虽然这只是一个简单的例子，它展示了如何不通过第三方的模块就能操作 `process.argv`。因为它就是一个 JavaScript 数组，可以非常容易地使用它，你可以轻而易举地通过如 `map`、`forEach` 以及 `slice` 来处理它。

复杂参数

对于比较复杂的程序，可以使用参数解析模块。两个最受欢迎的模块是 `optimist` (<https://npmjs.org/package/optimist>) 和 `commander` (<https://npmjs.org/package/commander>)。 `optimist` 将参数转换成一个对象来更容易地操作它。它还支持默认值、自动生成用例，以及简单的验证来确保必需的参数正确性。`commander` 和它有点不同：它通过链式 API 让你能够指定你的程序所能接受的参数。

好的 UNIX 程序在需要的时候处理参数，并且在退出的时候返回合适的状态码。下一节展示了如何以及什么时候使用 `process.exit` 来表示程序的成功、失败。

技巧 10 退出程序

Node 允许你在程序终止的时候给出一个退出码。

问题

你的 Node 程序需要在退出的时候指定退出码。

解决方案

使用 `process.exit()`。

讨论

无论在 Windows 还是 UNIX 中，退出码都是很重要的。其他的程序会通过检查退出码来确定一个程序是否执行成功。当你写的程序是一个大系统的一部分时，这显得尤其重要，并且对之后的维护以及调试都有帮助。

Node 程序默认返回 0 的退出状态。这意味着程序正常终止。任何的非 0 状态码被认为是一个错误。在 UNIX 中，这个状态码通常可以通过 `$?` 在 shell 中获取。在 Windows 中可以通过 `%errorlevel%` 获取。

例子 2.15 修改了例子 2.14 的代码, 使得程序在使用 -r 没有指定文件名的时候以相应的状态码退出程序。

例子 2.15 返回正确的退出状态码

```
var args = {
  '-h': displayHelp,
  '-r': readFile
};

function displayHelp() {
  console.log('Argument processor:', args);
}

function readFile(file) {
  if (file && file.length) {
    console.log('Reading:', file);
    require('fs').createReadStream(file).pipe(process.stdout);
  } else {
    console.error('A file must be provided with the -r option');
    process.exit(1);
  }
}

if (process.argv.length > 0) {
  process.argv.forEach(function(arg, index) {
    args[arg].apply(this, process.argv.slice(index + 1));
  });
}
```

❶ `Console.error` 和 `process.exit` 都是用来正确地指示发生了错误的情况。

在运行了例子 2.15 的代码之后, 输入 `echo $?` 在 UNIX 终端中将显示 1。并且要注意的是 `console.error` ❶ 被用来输出一个错误消息。这将导致这个消息被写进 `process.stderr`, 这使得这个脚本的用户可以简单地将错误重定向到其他地方。

退出码的特殊意义

在 *Advanced Bash-Scripting Guide* (<http://tldp.org/LDP/abs/html/index.html>) 中, 有一个章节主要讲解了退出码的特殊意义。它试图来标准化错误码, 尽管在脚本语言中没有一个标准的错误码列表, 0 以外的数字都表示错误发生。

因为许多 Node 程序是异步执行的，有些时候你需要显式地调用 `process.exit` 或者关闭 I/O 链接来使 Node 程序能优雅地退出。比如，在使用 Mongoose 的时候需要在程序退出之前调用 `mongoose.connection.close()`。

你可能需要跟踪在等待中的异步操作来决定什么时候安全地调用 `mongoose.connection.close()`，大多数开发者使用一个简单的计数器，在异步操作开始的时候增加它，在结束的时候减少它。一旦它达到 0，那么就能安全地关闭连接。

另外一个在开发正确程序时的重要因素是如何处理型号量。接下来你会学习到 Node 如何实现型号量的处理以及何时使用它们。

技巧 11 响应信号量

Node 的程序可以响应其他进程发出的信号。

问题

你需要响应其他进程发出的信号。

解决方案

使用发给 `process` 对象的信号事件。

讨论

大多数现代的操作系统通过信号把简单的消息发给一个程序。处理信号的程序通常运行在一个程序的后台，因为这可能是与它们通信的唯一办法。它们在你写的程序中可能很有用。比如在一个 web 应用中，当它收到 `SIGTERM` 信号时可以干净地关闭与数据库的连接。

`process` 对象是一个 `EventEmitter` 对象，这意味着你可以对它添加监听器。在 UNIX 系统中对一个 POSIX 信号添加一个监听器就可以了，你可以通过 `man sigaction` 来查看所有的信号。

对信号的监听可以用来满足 UNIX 程序期待的行为。比如，许多服务器以及进程守护程序在收到 `SIGHUP` 信号时会重新加载配置文件。下面的例子展示了如何添加一个 `SIGHUP` 的监听器。

例子 2.16 对 POSIX 信号添加一个监听器

```
process.stdin.resume();
process.on('SIGHUP', function () {
  console.log('Reloading configuration...');
});
```

❶

❷

```
console.log('PID:', process.pid);
```

③

① 从标准输入流中读取，这样程序会一直执行，直到按下 CTRL-C 或者进程被杀死。

② 对 SIGHUP 信号绑定一个监听器。

③ 显示 PID，你可以用它来终止进程。

在对标准输入进行任何操作之前，应该调用 `resume` ① 来防止 Node 直接退出，接下来对 `process` 对象添加一个 SIGHUP 事件 ②。最后显示当前进程的 PID ③。

一旦例子 2.16 的程序运行，它将显示进程的 PID。PID 可以用来阻止进程。比如 `kill -HUP 94962` 会将 HUP 信号发送给进程 94962。假如你发送了另外一个信号，或者输入 `kill 94962`，那么这个进程将会退出。

重要的是要知道，信号可以从任意的进程发送给其他进程，你的 Node 进程可以通过 `process.kill(pid, [signal])` 向另外一个进程发送信号，在这里，`kill` 不是意味着进程将被杀死，而是发送了一个信号。这个方法名是根据在 `signal.h` 中的 C 语言标准函数定义的。

图 2.3 展示了在操作系统中，任何进程如何发起信号、如何通过你的 Node 进程接收。

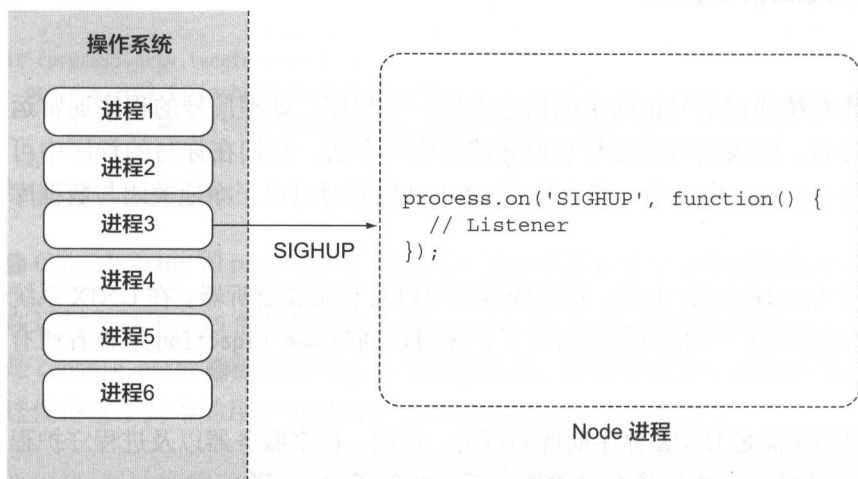


图 2.3 来自进程的信号量，使用事件监听器来处理

在你的 Node 程序中你不需要对信号进行响应，但是，如果你在写一个长时间运行的网络服务，那么信号处理会相当有用。对 SIGHUP 这样的信号的支持将会使你的程序很自然地集成到现有的系统中。

Node 的一个很大的吸引力是它的异步接口以及非阻塞式的 I/O 特性。有些时候我们希望来模拟这个行为，例如在自动化测试中，或者强制让代码执行。下一节我们会看到 Node 如何通过实现 JavaScript 的 timer 来支持这个功能。

2.4 使用 timer 延迟执行

Node 实现了 JavaScript 的计时器的功能——`setTimeout`、`setInterval`、`clearTimeout`，以及 `clearInterval`，这些方法全局可用。尽管 Mozilla 定义它们是 JavaScript 的一部分，但是它们没有被定义在 ECMAScript 标准中。相反，它们是 HTML DOM Level 0 标准的一部分。

技巧 12 通过 `setTimeout` 延迟执行函数

可以通过 Node 的 `setTimeout` 全局方法延迟执行一段代码。

问题

你想延迟执行一个函数。

解决方案

使用 `setTimeout`，并且在需要的时候使用 `Function.prototype.bind`。

讨论

一个最简单的 `setTimeout` 的用法是：将一个函数传入它，在 1 秒后执行。

```
setTimeout(function() {  
  console.log('Hello from the past!');  
}, 1000);
```

这看上去简单，但你会看到在测试中经常会通过这种手段来模拟真实的异步操作。Node 对 JavaScript 计时器的支持通常是为了这些情况。

方法可以被很容易地通过 `Function.prototype.bind` 传入 `setTimeout`。这可以被用来将第一个参数绑定为 `this`，或是更多见的这个方法隶属的对象。下面的例子展示了 `bind` 是如何使用的。

例子 2.17 `setTimeout` 与 `Function.prototype.bind`

```
function Bomb() {  
  this.message = 'Boom!';  
}
```

```
Bomb.prototype.explode = function() {  
  console.log(this.message);  
};  
  
var bomb = new Bomb();  
  
setTimeout(bomb.explode.bind(bomb), 1000);
```

❶ 通过调用 `.bind` 可以确保这个方法绑定到正确的对象上，这样它可以访问这个对象的内部属性。

绑定确保了该方法内的代码可以访问对象的内部属性，否则，`setTimeout` 会导致与 `this` 绑定到全局对象运行。绑定方法相比创建一个新的匿名函数更加具有可读性。

要取消将被执行的函数，需要保存 `setTimeout` 函数执行返回的 `timeoutId`，然后通过调用 `clearTimeout(timeoutId)` ❶ 来取消。下面的例子展示了 `clearTimeout`。

例子 2.18 使用 `clearTimeout` 来阻止定时函数执行

```
function Bomb() {  
  this.message = 'Boom!';  
}  
  
Bomb.prototype.explode = function() {  
  console.log(this.message);  
};  
  
var bomb = new Bomb();  
  
var timeoutId = setTimeout(bomb.explode.bind(bomb), 1000);  
  
clearTimeout(timeoutId);
```

❶ 调用 `clearTimeout` 来在运行中阻止 `bomb` 的 `explode`。

回调何时执行?

尽管你可以指定在某时执行一个回调，但 Node 没有那么准确。它能确保回调在一定的时间执行，但可能会有些延迟。

除了延迟执行，你还可以定时调用函数。下面的技巧描述了如何通过 `setInterval` 达到这个目的。

技巧 13 通过定时器定时调用回调函数

Node 还可以像调用 `setTimeout` 那样通过 `setInterval` 在固定时间间隔内执行一个回调函数。

问题

你想在一个固定时间间隔运行回调。

解决方案

通过 `setTimeout`，并且使用 `clearInterval` 来终止定时器。

讨论

`setInterval` 方法在浏览器端中的使用已经有多年的历史了，而它在 Node 中的行为和它的客户端老朋友差不多。回调将在指定的延迟后执行，并将在事件循环中运行（以及任何调用 `setImmediate`，在技巧 14 详细介绍）。

接下来的例子显示了如何结合 `setInterval` 和 `setTimeout` 使两个函数序列执行。

例子 2.19 使用 `setInterval` 和 `setTimeout`

```
function tick() {  
  console.log('tick:', Date.now());  
}
```

```
function tock() {  
  console.log('tock:', Date.now());  
}
```

```
setInterval(tick, 1000);
```

```
setTimeout(function() {  
  setInterval(tock, 1000);  
, 500);
```

❶

❶ 在第一个之后运行另外一个 `setInterval`。

`setInterval` 方法返回一个指向计时器的引用，它可以通过调用 `clearInterval` 来终止计时器。例子 2.19 通过调用 `setTimeout` ❶ 来触发第二个计时器在 500 秒后开始执行。

因为 `setInterval` 会阻止程序退出，确实有这种情况——你希望在没有其他操作的时候退出程序。比如说，你正在执行一个程序，这个程序应该在一个复杂的操作结束后退出，而且你希望通过 `setInterval` 定期地监视这个程序。一旦这个复杂的操作结束，你就不希望再监视这个程序。

除了调用 `clearInterval`, Node 0.10 版本中还允许你在这个复杂操作结束之前执行 `timer-Ref.unref()`。这意味着你可以和一些操作同时使用 `setTimeout` 或者 `setInterval`, 而不用在操作执行后通知 `timer` 它们结束了。

例子 2.20 通过 `setTimeout` 来模拟一个长时间运行的操作, 这个操作执行的同时还会显示进程的内存使用情况。一旦 `timeout` 到时间了, 这个程序将自动退出而不用再去执行 `clearTimeout`。

例子 2.20 让计时器持续运行直到程序退出

```
function monitor() {  
  console.log(process.memoryUsage());  
}  
  
var id = setInterval(monitor, 1000);  
id.unref();  
  
setTimeout(function() {  
  console.log('Done!');  
}, 5000);
```

❶ 在程序完成长时间的操作后, 告知 Node 停止定时器。

有时候没有合适的地方来调用 `clearInterval`, 这将变得特别有用。

一旦你掌握了 `timer`, 你将会碰到一些情况, 需要在一个尽可能短的延迟后执行一个回调。使用 `setTimeout` 为零的延迟不是最佳的解决方案, 即使它似乎是显而易见的策略。在下一个技巧中, 你将会看到如何正确地通过 `process.nextTick` 来做到这点。

技巧 14 安全的操作异步接口

有时你想略微延迟一下操作。在传统的 JavaScript 中, 或许通过 `setTimeout` 执行一个很小的延迟是可接受的。然而 Node 提供了一个更有效的方案, `process.nextTick`。

问题

你想写一个方法返回一个 `EventEmitter` 的实例, 或者允许一个回调仅在有些时候调用一个异步的接口, 而不是所有时候。

解决方案

使用 `process.nextTick` 来包装一个同步操作。

讨论

`process.nextTick` 方法允许你把一个回调放在下一次事件轮询队列的头上。这意味着它可以用来延迟执行，其结果是它比使用 `setTimeout` 更有效率。

很难想象为什么这是有用的。例子 2.21 展示了返回一个 `EventEmitter` 实例的方法。这个想法是提供一个基于事件的接口，允许调用者订阅这个事件在内部执行异步方法。

例子 2.21 错误地通过事件触发异步方法

```
var EventEmitter = require('events').EventEmitter;

function complexOperations() {
  var events = new EventEmitter();

  events.emit('success'); 1((callout-globals-nexttick-1))

  return events;
}

complexOperations().on('success', function() {
  console.log('success!');
});
```

❶ 这是一个在异步回调外触发的事件。

运行这个例子将不会在最后触发 `success` 事件❶。为什么呢？这是因为这个事件在监听器订阅之前就已经触发了。通常，一个事件会在一个异步的操作中触发，但有时候也会提早触发事件，比如在验证入参的时候发现有错误，那么 `error` 事件将被触发。

要纠正这个小错误，可以把这段代码包裹进 `process.nextTick` 中。下面的例子返回一个 `EventEmitter` 实例的方法，并在这个方法中触发了一个事件。

例子 2.22 在 `process.nextTick` 中触发事件

```
var EventEmitter = require('events').EventEmitter;

function complexOperations() {
  var events = new EventEmitter();

  process.nextTick(function() {
    events.emit('success');
  });

  return events;
}
```

```
}  
  
complexOperations().on('success', function() {  
  console.log('success!');  
});
```

❶ 这个事件现在会在监听器准备好后被触发。

Node 在文档中建议, 接口要么是同步的, 要么是异步的, 这表示如果你有一个方法接受一个回调, 并可能异步地调用它, 那么你也应该在同步的情况下通过 `process.nextTick` 来执行它, 这样可以确保执行的顺序性。

例子 2.23 异步地从磁盘上读取一个文件。一旦读取了文件, 它会在内存中缓存这个文件。之后的调用会直接返回缓存内容。当返回缓存时, 通过调用 `process.nextTick` 来使接口异步地执行。这确保了在终端中的输出顺序是正确的。

例子 2.23 创建一个始终异步的 API

```
var EventEmitter = require('events').EventEmitter;  
var fs = require('fs');  
var content;
```

```
function readFileSyncIfRequired(cb) {  
  if (!content) {  
    fs.readFile(__filename, 'utf8', function(err, data) {  
      content = data;  
      console.log('readFileSyncIfRequired: read');  
      cb(err, content);  
    });  
  } else {  
    process.nextTick(function() {  
      console.log('readFileSyncIfRequired: cached');  
      cb(null, content);  
    });  
  }  
}
```

```
readFileSyncIfRequired(function(err, data) {  
  console.log('1. Length:', data.length);  
  
  readFileSyncIfRequired(function(err, data2) {  
    console.log('2. Length:', data2.length);  
  });  
});
```

```
console.log('Reading file again...');  
});
```

```
console.log('Reading file...');
```

- ❶ 如果内容还没被读进内存，那么异步读取它。
- ❷ 如果内容已经读取好了，传递缓存的版本给回调，但是第一次使用 `process.nextTick` 来确保回调过一会儿才执行。
- ❸ 随后调用异步的操作来确保行为和预期中一样。

这个例子中，通过 `fs.readFile` 把一个文件读取到缓存中❶，然后在后面的每次调用中返回这个文件的副本❷。它被封装在一个进程中多次执行❸，所以你可以比较异步的文件操作与 `process.nextTick` 的不同。

可视化事件轮询：setImmediate 和 process.maxTickDepth

`setImmediate` 以及 `clearImmediate` 全局方法接受一个回调参数和可选的参数，它会在下一次 I/O 事件后并在 `setTimeout` 以及 `setInterval` 之前执行。

通过这个方法添加的回调函数被推入队列中，而且在每次轮询时执行一个回调。这和 `process.nextTick` 不同，导致 `process.maxTickDepth` 回调在每次轮询时都会执行。

传入 `process.nextTick` 的回调通常在当前事件轮询结束后执行。可以被安全执行的回调数量被 `process.maxTickDepth` 控制，默认的是 1000，以允许 I/O 操作可以继续被处理。

图 2.4 展示了在一次事件轮询中各个 timer 函数的位置。

当你在创建你自己的异步函数或者对象时，通过 `process.nextTick` 来确保函数的行为是一致的、可预测的。

Node 中实现的基于浏览器的定时器可以在事件轮询中正常执行。尽管它们主要是用来测试异步代码的，深入了解 `setTimeout`、`setImmediate` 以及 `process.nextTick` 何时执行需要掌握事件轮询。

2.5 总结

在这一章中，你已经看到了 Node 中很多强大的内置的功能，这些功能不需要加载一个模块。下一次你想组织相关的模块时，你可以创建一个 `index.js` 文件，就如技巧 3 中提到的那样。假如你需要读取标准输入，你可以通过 `process` 对象的 `stdin` 属性（技巧 5）。

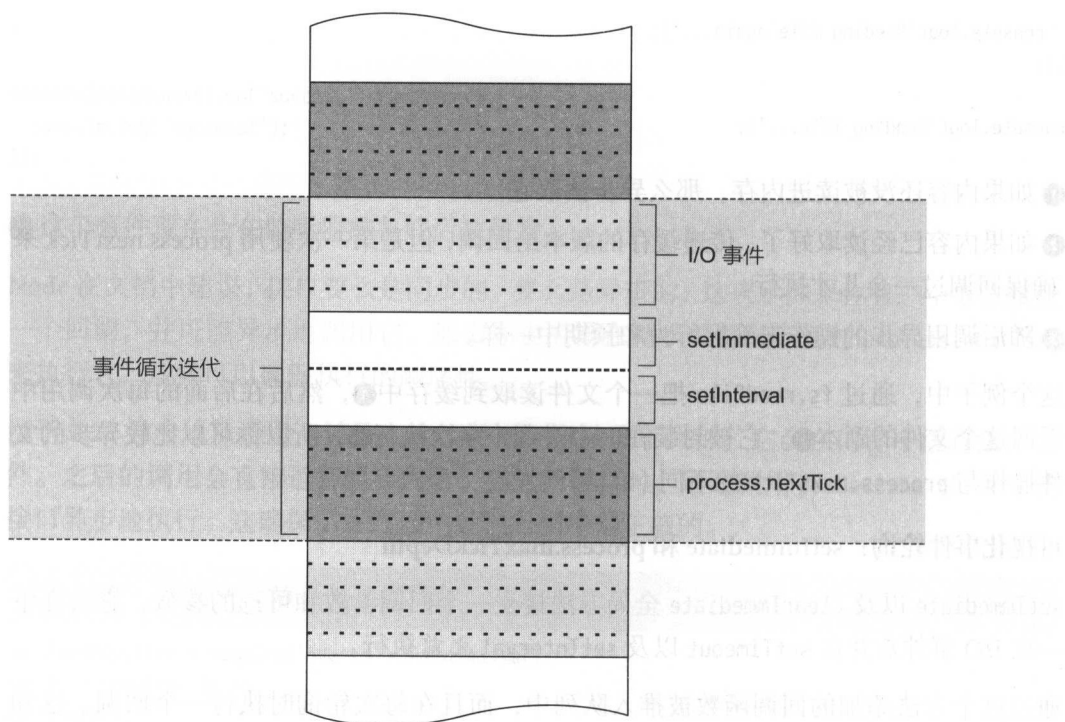


图 2.4 事件循环中 nextTick 的时序安排

除了 `process` 对象，还有经常被忽视的 `console` 对象，它可以帮助你调试并且维护程序（技巧 6）。

在下一章中，你将学习到 `buffer`。`buffer` 非常适合处理二进制数据，这通常被认为是 JavaScript 的弱点。`Buffer` 还加强了 Node 的强大特性，例如 `Stream`。

Buffers：使用比特、字节以及编码

本章概要

- 介绍 Buffer 数据类型
- 修改数据编码
- 二进制文件转化为 JSON 格式
- 创建自定义的二进制协议

历史上 JavaScript 对二进制支持欠佳。通常情况下，解析二进制数据涉及到从字符串中提取所需要的数据的多种技巧。随着项目的进展，Node 开发者需要面对的其中一个问题是，JavaScript 中没有提供一个比较好的方式去处理原始的内存数据。出于对性能的考虑，所有关于原始内存数据的处理都在 Buffer 数据类型中。

Buffers 是代表原始堆的分配额的数据类型，在 JavaScript 中以类数组的方式来使用。在全局可用，不需要 require，可以将其视为是 JavaScript 的另外一种类型（就像 String 和 Number 一样）：

```
var buf = new Buffer(255);  
buf[0] = 23;
```

❶
❷

- ❶ 分配 255 个字节。
- ❷ 第一个字节写入整型数据 23。

如果对处理二进制不是特别熟悉也不用担心，这个章节的内容不但针对新手入门，也是对有一定基础的开发人员的进阶。我们将会从简单到高级深入浅出地介绍。

- 将 Buffer 转化为不同的编码。
- 使用 Buffer 的 API 把一个二进制文件转换为 JSON 格式。
- 使用自定义的二进制协议来编码解码。

我们先来看修改 Buffer 的编码。

3.1 修改数据编码

如果没提供编码格式，那么文件操作以及很多的网络操作就会将数据作为 Buffer 类型返回，就拿 `fs.readFile` 作为例子：

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  Buffer.isBuffer(buf); // true
});
```

❶ 如果是 Buffer 类型则会返回 true。

但大部分情况下你是已经知道了文件的编码的，比较有效的方式是直接获得对应编码的字符串。在这一章节我们看一下 Buffer 和其他格式之间是如何相互转换的。

技巧 15 Buffer 转换为其他格式

默认情况下，没有指定编码格式，Node 的一些核心 API 都会返回 Buffer 数据。Buffer 很容易转换成其他格式，在这个技巧中我们来看一下这是如何转换的。

问题

你需要把一个 Buffer 转换为文本类型。

解决方案

Buffer API 提供了把 Buffer 转换为字符串的方法。

讨论

假设我们有一个内容为纯文本的文件，文件名为 `names.txt`，该文件每一行都有一个人的名称：

```
Janet
Wookie
```


Alex
Marc

我们使用文件系统（fs）的 API 来加载这个文件，默认情况下，我们会得到一个 Buffer（buf）。

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  console.log(buf);
});
```

上边的代码输出结果是一串八位字节组（16 进制编码）：

```
<Buffer 4a 61 6e 65 74 0a 57 6f 6f 6b 69 65 0a 41 6c 65 78 0a
      4d 61 72 63 0a>
```

这并不是很有用，因为我们已经知道这个文件的编码格式。Buffer 类型提供了 toString 这个方法把数据转换为 UTF-8 编码的字符串：

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  console.log(buf.toString());
});
```

❶ toString 默认把数据转换为 UTF-8 格式的字符串。

现在可以获得和文件内容一样的输出：

Janet
Wookie
Alex
Marc

但是，当我们知道这个数据仅仅包含了 ASCII 字符时，¹我们也可以把编码改为 ASCII 来提升性能。为了实现这个，我们需要提供编码类型作为 toString 的第一个参数：

```
var fs = require('fs');
fs.readFile('./names.txt', function (er, buf) {
  console.log(buf.toString('ascii'));
});
```

❶ toString 接受第一个参数作为编码类型。

¹ 参见 <http://en.wikipedia.org/wiki/ASCII>。

Buffer API 提供了其他编码类型, 类似 utf16le、base64 和 hex 等, 你可以从 Buffer API 在线文档获取更多相关信息。²

技巧 16 使用 Buffers 来修改字符串编码

除了把 Buffers 转换为字符串, 你也可以利用 Buffer 来进行字符串的编码格式转换。

问题

你需要把一个字符串的编码格式转换成另外一种。

解决方案

Node 中的 Buffer API 提供了转换字符串编码的方法。

讨论

例子 1: 创建基本的验证头部信息

有时候, 创建一个字符串数据后修改它的编码格式是很有用的。例如, 当你需要从一个使用基础验证³的服务器请求数据时, 你需要发送使用 Base64 编码的用户名和密码:

```
Authorization: Basic am9obm55OmMtYmFk
```

❶

❶ am9obm55OmMtYmFk 是编码好的。

在进行 Base64 编码前, 基础验证需要把用户名和密码拼接到一起, 用 : 分隔开来, 例如, 我们使用 johnny 作为用户名, 而 c-bad 作为密码:

```
var user = 'johnny';  
var pass = 'c-bad';  
var authstring = user + ':' + pass;
```

❶

❶ 用户名和密码用: 分隔开。

现在我们需要把这个字符串转换为 Buffer, 然后修改它的编码格式。Buffers 可以按照字节数来分配, 如之前我们看过的简单传入数字的创建方式 (例: new Buffer(255))。Buffer 也可以通过传入字符串数据来创建:

```
var buf = new Buffer(authstring);
```

❶

❶ 字符串数据转换为 Buffer。

² 参见 <http://nodejs.org/api/buffer.html>。

³ 参见 http://en.wikipedia.org/wiki/Basic_access_authentication。

指定编码

当使用字符串来创建 Buffer 时, 默认为 UTF-8 字符串。但是当我们需要指定数据的编码时, 可以传入第二个可选的参数来表示对应的编码:

```
new Buffer('am9obm55OmMtYmFk', 'base64')
```

现在有一个 Buffer 来保存数据, 可以使用 `toString('base64')` 方法来把它转换为 Base64 编码的字符串:

```
var encoded = buf.toString('base64');
```

❶ 结果为 `am9obm55OmMtYmFk`。

我们的代码可以写得更加紧凑点, 在使用 `new` 创建 Buffer 实例后可以直接调用实例的方法:

```
var encoded = Buffer(user + ':' + pass).toString('base64');
```

例子 2: 处理 data URIs

Data URIs⁴是说明 Buffer API 非常有用的另外一个例子。Data URIs 允许一个资源以行内编码的形式存在于 web 页面中, 只需要遵从以下的格式:

```
data:[MIME-type][:charset=<encoding>[:base64],<data>
```

例如, 一个猴子的 png 图片可以以 data URI 的方式展现:

```
data:image/png;base64,iVBORw0KGgoAAAANSUHEUgAAACsAAAAoCAYAAABny...
```

当在浏览器时, 这个 data URI 会呈现出图 3.1 中的小猴子。

我们看一下如何使用 Buffer API 来创建 data URI。以上述的小猴子为例, 我们使用 png 图片格式, 其 MIME 类型为 `image/png`:

```
var mime = 'image/png';
```



图 3.1 浏览器中的 Data URI 以图片的形式显示一只猴子

二进制文件可以使用 Base64 编码以 data URIs 的方式呈现, 所以我们先声明一个编码变量:

```
var encoding = 'base64';
```

⁴参见 http://en.wikipedia.org/wiki/Data_URI_scheme。

使用 `mime` 和编码格式, 我们可以先构建 `data URI` 的开头部分:

```
var mime = 'image/png';
var encoding = 'base64';
var uri = 'data:' + mime + ';' + encoding + ',';
```

接下来需要添加主要的图片数据, 我们使用 `fs.readFileSync` 同步地读取文件数据并且直接返回。`fs.readFileSync` 将返回一个 `Buffer`, 我们把它转换为 `base64` 编码的字符串:

```
var encoding = 'base64';
var data = fs.readFileSync('./monkey.png').toString(encoding);
```

我们将上述的代码组合起来, 便可以创建一个程序用于输出 `data URI`:

```
var fs = require('fs');
var mime = 'image/png';
var encoding = 'base64';
var data = fs.readFileSync('./monkey.png').toString(encoding);
var uri = 'data:' + mime + ';' + encoding + ',' + data;
console.log(uri);
```

❶ 引入 `fs` 模块来使用 `fs.readFileSync`。

❷ 构建 `data URI`。

❸ 输出 `data URI`。

该程序的输出会是这样的:

```
data:image/png;base64,iVBORw0KGgoAAAANSUHEUgAAACsAAAAoCAYAAABny...
```

我们可以再琢磨一下。如果我们有 `data URI` 并且需要把它写入一个文件, 应该怎么办? 同样地, 以小猴子为例子。我们使用 `split` 方法来分割数组以获取需要的数据:⁵

```
var uri = 'data:image/png;base64,iVBORw0KGgoAAAANSUHEUgAAACsAAAAo...';
var data = uri.split(',')[1];
```

使用字符串数据来创建一个 `Buffer` 并且指定它的编码:

```
var buf = Buffer(data, 'base64');
```

接下来, 使用 `fs.writeFileSync` 来把数据同步地写入磁盘, 传入两个参数, 文件名称和存放数据的 `Buffer`:

```
fs.writeFileSync('./secondmonkey.png', buf);
```

⁵这并不是对所有的 `data URIs` 都能够适用, 逗号可能在其他位置也会出现。

将上述的代码放到一起：

```
var fs = require('fs');  
var uri = 'data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAACsAAAAo...';  
var data = uri.split(',')[1];  
var buf = Buffer(data, 'base64');  
fs.writeFileSync('./secondmonkey.png', buf);
```

❶ 引入 fs 模块来使用 fs.writeFileSync。

我们使用默认的图片预览器打开文件时，可以看到图 3.2 中的小猴子。

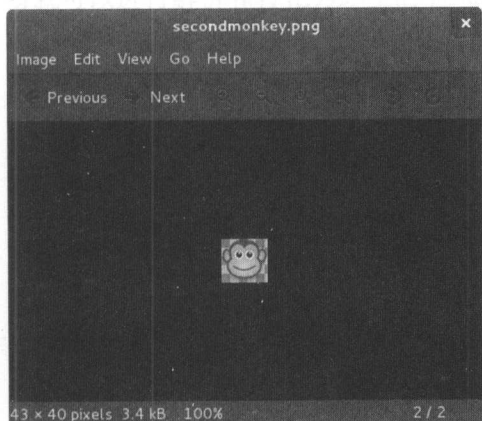


图 3.2 从 data URI 中生成 secondmonkey.png 文件

大部分情况下，当你使用 Node 中的 Buffer 对象时，只是需要把它转换为其他格式，有时候是更改其编码格式。但是，有时候你可能需要处理二进制文件内容，Buffer API 已经提供了很丰富的工具来帮助你完成任务，我们会在下一章深入了解。

3.2 二进制文件转换为 JSON

处理二进制数据就像解决一个难题一样。通过阅读数据含义的说明来获取线索，然后重新整理，把这些数据转化为程序可用的东西。

技巧 17 使用 Buffer 来转换原始数据

怎样在 Node 程序中利用二进制格式来做一些有意义的事情？在这个技巧中，我们会进行详细介绍如何处理二进制文件，并将其转换为常见的 JSON 格式文件。

问题

你需要把一个二进制文件转换为更加有用的数据格式。

解决方案

Node API 在 JavaScript 基础上提供了 Buffer 的 API，来读取原始的二进制数据，以及提供了一些工具来帮助我们更加简单地处理二进制数据。

讨论

关于我们例子的文件转换流程，可以参考图 3.3。

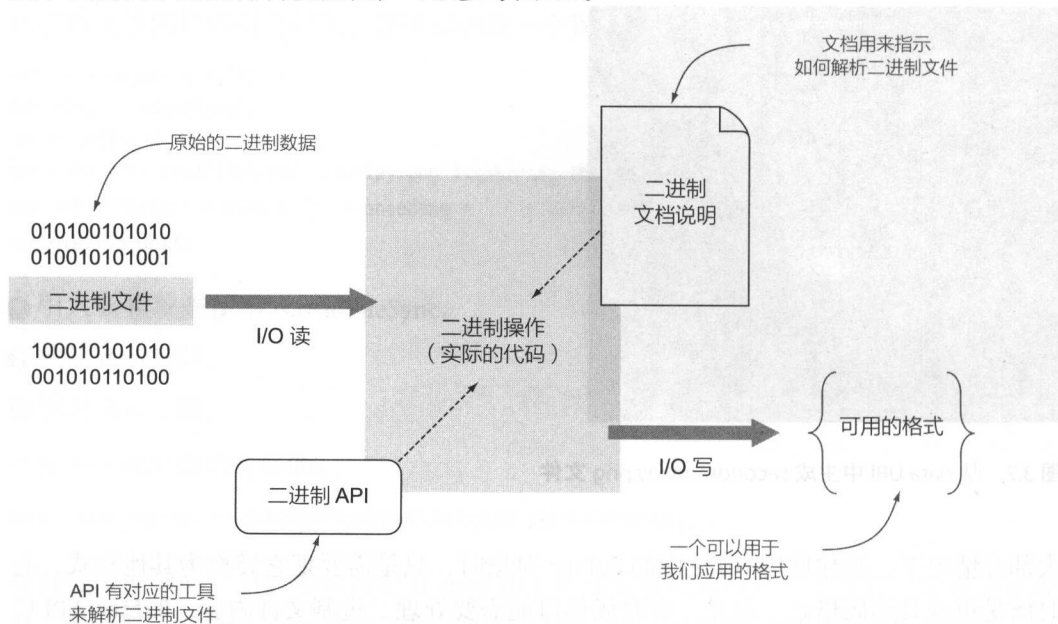


图 3.3 将二进制数据转换成更可用的格式

使用特定的规范作为指导，以便读取、处理，以及将二进制数据转换为更加有用的格式，二进制的 API 则是作为一种处理数据转换的方式。这不是二进制数据的唯一用途。例如，你还可以基于二进制协议来进行数据的来回传输。

我们的技巧介绍中，将会来处理 DBase 5.0 (.dbf) 的二进制文件。这个格式看起来可能有点晦涩，但是它是一种流行的数据库文件格式，在地理空间领域的数据处理中大量使用。你可以把它当作一种简单的 Excel 表。我们将要处理的样本存放在 `buffers/world.dbf`。这个文件包含了世界上多个国家的地理空间信息。不幸的是，在编辑器中打开后，你无法看到有用的信息。

为什么我们要深入介绍可能都不会用到的二进制格式的处理？

我们挑选了多个二进制的格式, 相比其他的, DBase 5.0 是一个比较好的格式, 可以用来帮助我们了解处理二进制文件的多种不同的方式。同时, 对于很多有 web 开发背景的开发来说, 对二进制格式可能都不是特别熟悉, 需要花费一些时间去了解如何处理二进制相关的规范。如果对这一方面比较熟悉, 可以跳过这一部分。

我们需要在 Node 程序中使用到的 JSON 格式, 是一种非常好的选择, 因为 JavaScript 本身就可以进行解析和转化为原生的 JavaScript 对象。在图 3.4 中将会说明。

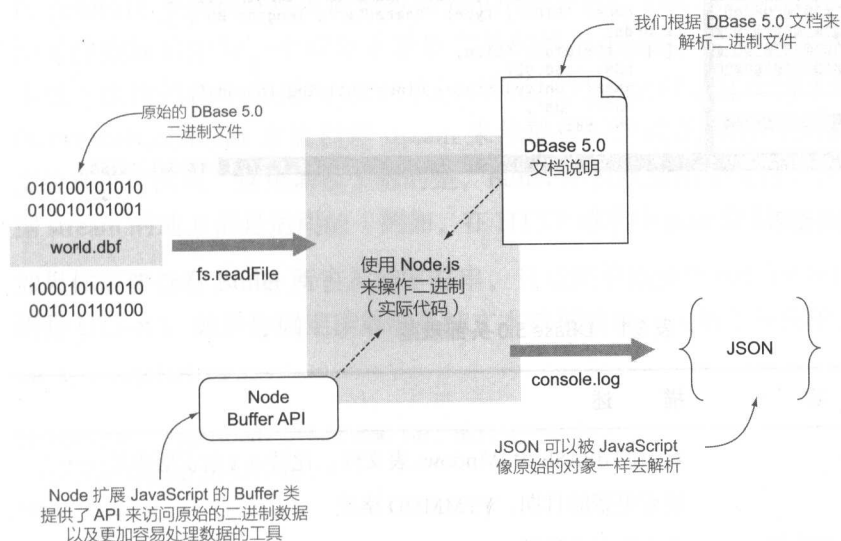


图 3.4 使用文件系统的 API 读取二进制文件, 然后用 Buffer API 转换为更加易用的 JSON 格式

图 3.5 展示了我们需要实现的转化: 左侧是编辑器中打开的二进制数据, 右侧是转换后的 JSON 格式。

头部信息

我们开始处理这个问题前, 我们需要调查一下, 找出要处理的二进制文件格式的规范。在这个例子中, 通过网上的搜索引擎可以找到一些可以参考的规范。对于 DBase 5.0, 在这个例子中使用到的主要规范可以参考这个网址: <http://mng.bz/i7K4>。

规范中的第一部分主要是头部信息。很多二进制文件会使用头部信息来存储文件相关的基础信息。表 3.1 展示了 DBase 5.0 规范中的信息:

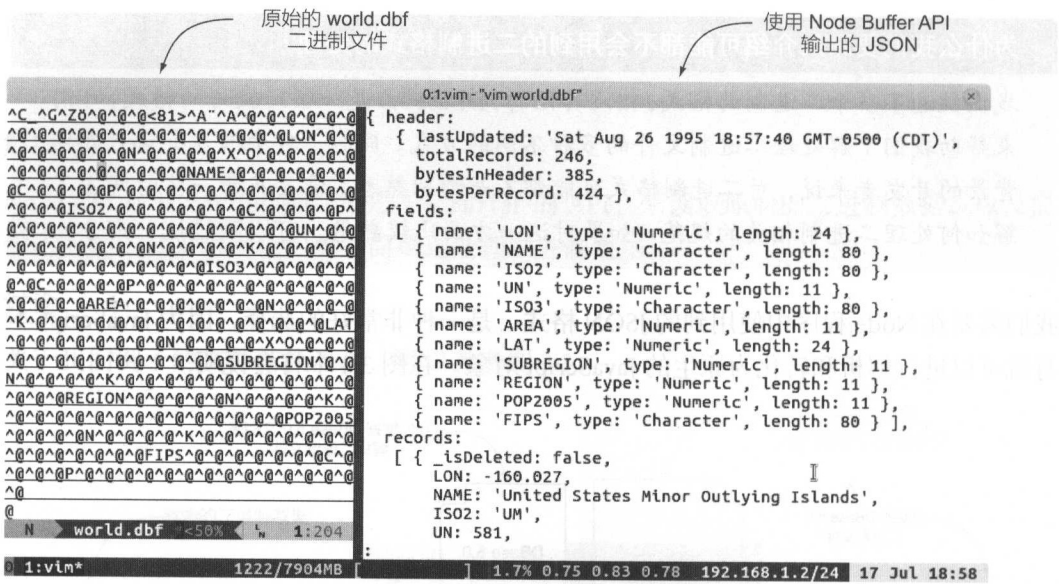


图 3.5 我们转换的最终结果

表 3.1 DBase 5.0 头部规范

Byte	内 容	描 述
0	1 byte	有效的 dBASE 为 Windows 表文件；比特 0-2 指示版本号……
1-3	3 byte	最后更新的日期，YYMMDD 格式
4-7	32-bit 的数字	表中的记录数量
8-9	16-bit 的数字	头部的字节数
10-11	16-bit 的数字	记录部分的字节数
...
32-n each	32 byte	字段描述
n+1	1 byte	字段分割符

我们先看一下第一行：

Byte	内 容	描 述
0	1 byte	有效的 dBASE 为 Windows 表文件；比特 0-2 指示版本号……

这一行告诉我们最开始的一个字节包含了描述中提到的相关信息。那么我们如何取得最开始的一个字节的数据。幸运的是，使用 `buffer` 处理起来很简单。

Node 中，除非你为读取的数据指定编码，不然结果都会返回一个 Node Buffer 对象，例如：

```
var fs = require('fs');

fs.readFile('./world.dbf', function (er, buf) {
  Buffer.isBuffer(buf); // true
});
```

`fs.readFile` 不是唯一可以获取数据 `buffer` 的方法，为简单起见，我们使用这个方法以便文件读取后作为一个对象来获取完整的数据 `Buffer`。这个方法不适用于比较大的，不想一次性把整个数据内容加载进内存的二进制文件。这个例子中，你也可以使用 `fs.createReadStream` 方法创建 `stream` 来传输数据，或者使用 `fs.read` 把文件拆分成许多块来进行读取。这里需要了解的是，`Buffer` 不仅仅适用于文件，任何你可以获取到数据 `stream` 的地方都是适用的（例如，在 HTTP 请求中 `post` 数据）。

如果你需要查看 `Buffer` 所表示的字符串，只是简单地使用 `buf.toString()` 便可（默认编码是 UTF-8）。如果你确定读取的是纯文本数据的画，这将十分简单：

```
var fs = require('fs');

fs.readFile('./world.dbf', function (er, buf) {
  console.log(buf.toString());
});
```

❶

❶ 默认返回 UTF-8 编码的字符串。

在我们的例子中，`buf.toString()` 和在文本编辑器中直接打开 `world.dbf` 文件的结果一样糟糕。我们需要先把数据处理成有意义的格式。

注意

直到现在，经常可以看到 `buf` 变量，这个是 `Buffer` 类的一个实例，是可以使用的 Node API 的一部分。

关于上边的表格，我们谈及，在 Node 中，从 `Buffer` 获取索引为 0 的数据和 JavaScript 中数组相关的操作非常相似，只是索引是指在内存中的字节位置。所以第 0 个字节位置便是 `buf[0]`。在 `Buffer` 语法中，`buf[0]` 即代表了第 0 位的八位字节，或无符号的 8 比特整数，或 8 比特的正整数，这都是同一个意思。

回到例子中，我们无须关注这个特殊字节存储的数据信息，来看看下一个字节的定义。

Byte	内 容	描 述
1-3	3 byte	最后更新的日期，YYMMDD 格式

最后更新的日期，是比较有用的信息。但是除了这个 3 字节数据和格式为 YYMMDD 的信息，并没有更加详细的说明。这意味着你可能无法在一个地方找到所需要的东西。我们可以在网络上搜索到以下信息：

每一个字节以二进制的方式包含了所要表示的数字。YY 表示以 1900 为基础的十位数和个位数，用于计算出确切的年份。YY 可能的取值范围为 0x00-0xFF，即 1900-2155 之间。^a

^a参见 http://www.dbase.com/Knowledgebase/INT/db7_file_fmt.htm。

该信息十分有用，我们可以在 Node 中进行解析：

```
var header = {};  
  
var date = new Date();  
date.setUTCFullYear(1900 + buf[1]);  
date.setUTCMonth(buf[2]);  
date.setUTCDate(buf[3]);  
header.lastUpdated = date.toUTCString();
```

❶ 结果是 “Sat Aug 26 1995 ...”

在这里我们使用 JavaScript 的 Date 对象，将其年份设置为 1900 加上从 buf[1] 中获取的数值。我们使用第 2、第 3 位数值分别作为月份和日期。由于 JSON 格式无法存储 Date 对象，我们将其转换为表示 UTC 日期的字符串。

我们停下来回顾一下。“Sat Aug 26 1995”便是通过解析 world.dbf 的二进制数据的一部分，获取到的一个 JavaScript 的字符串数据。接下来可以看到更多的例子。

Byte	内 容	描 述
4-7	32-bit 的数字	表中的记录数量

这一部分的定义给了我们两个线索。我们可以知道从第 4 个字节开始，是一个 32 比特的数值类型。我们可以确定这个数值一定是有效的，假设是一个正整数类型，或者无符号整数类型，都可以使用 Buffer API 的这个方法来获取到：

```
header.totalRecords = buf.readUInt32LE(4);
```

❶

❶ 结果是 246。

`buf.readUInt32LE` 可以从第 4 个字节开始，以小端格式读取一个无符号的 32 比特正整数，这正符合了文档规范的定义。

接下来两部分的定义很相似，除了是一个 16 比特的整型数值，如下：

Byte	内 容	描 述
8-9	16-bit 的数字	头部的字节数
10-11	16-bit 的数字	记录部分的字节数

下边是对应的代码：

```
header.bytesInHeader = buf.readUInt16LE(8);
```

❶

```
header.bytesPerRecord = buf.readUInt16LE(10);
```

❷

❶ 结果：385。

❷ 结果：424。

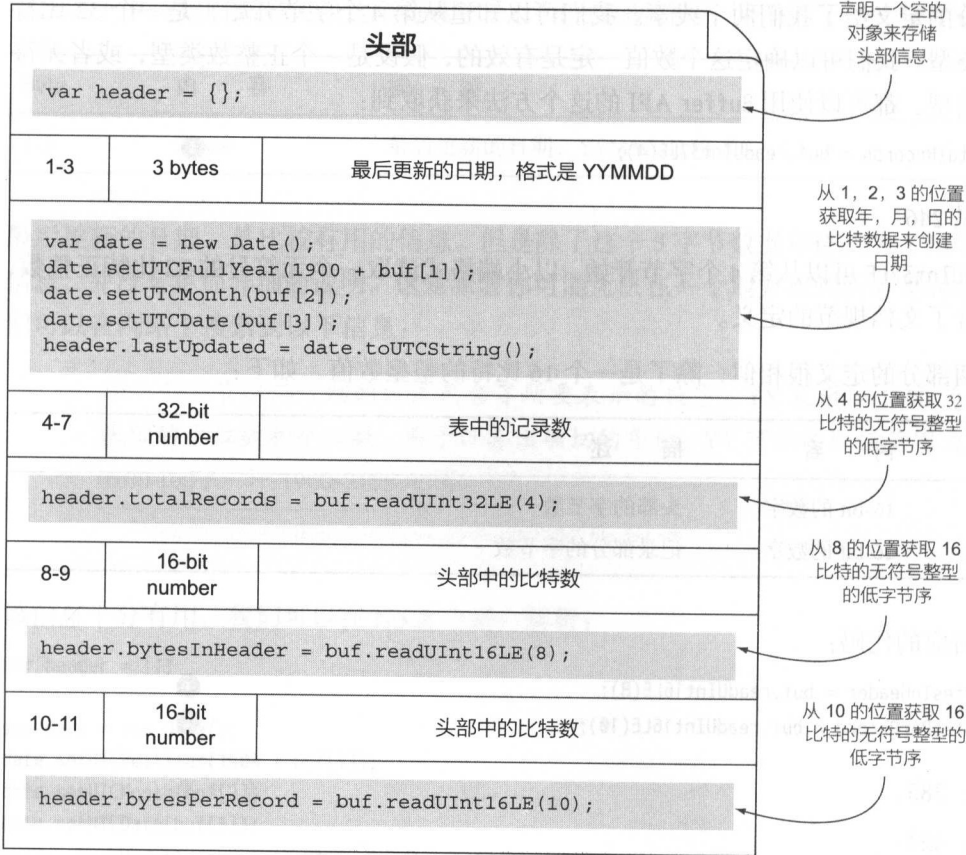
这一部分关于规范和头部信息的处理说明可以参考图 3.6。

字段描述数组

我们的例子中，在 `world.dbf` 文件的头部信息只有一个和数据格式关系密切的部分。它是所有字段的定义，包括了字段类型和名称信息，可以看下边的图表：

Byte	内 容	描 述
32-n each	32 byte	字段描述
n+1	1 byte	字段分割符

从这里我们可以看出，每一个字段的描述是以一个 32 字节的数据存储起来的。数据库可能包括了多个字段的数据，字段数据的结束是由图表中第二行定义中的单字节的终止符来进行标示的。我们可以写一个数据结构来处理：



转换的结果 →

```
header = {"lastUpdated": "Sat Aug 26 1995 ...",
"totalRecords": 246,
"bytesInHeader": 385,
"bytesPerRecord": 424}
```

图 3.6 头部: 使用 Node 的 Buffer API 将规范转换成代码

```
var fields = [];
var fieldOffset = 32;
var fieldTerminator = 0x0D;

while (buf[fieldOffset] !== fieldTerminator) {
  // 在这里解析每一个字段
  fieldOffset += 32;
}
```

① 代表 0Dh 的 JavaScript 字面量。

这里我们进行循环，每次处理 32 字节的数据，直到遇见以十六进制表示的字段终结符 (fieldTerminator)。

现在我们必须处理每一个字段描述的数据。规范中有另外一个图表来说明，例子中需要的相关信息可以在表 3.2 中看到：

表 3.2 DBase 5.0 字段描述

Byte	内 容	描 述
0-10	11 bytes	字段名
11	1 byte	字段类型
...
16	1 byte	字段长度

留意一下字节的索引是以 0 开始的，即使我们已经处理过文件的索引位置为 0 的字节。为了更方便理解规范中的说明，我们从每一个记录重新开始索引。Buffer 提供了 slice 方法来帮助我们处理：

```
var fields = [];  
var fieldOffset = 32;  
var fieldTerminator = 0x00;  
  
while (buf[fieldOffset] != fieldTerminator) {  
    var fieldBuf = buf.slice(fieldOffset, fieldOffset+32);  
    // 在这里解析每一个字段  
    fieldOffset += 32;  
}
```

buf.slice(start, end) 和标准的数组方法 slice 非常相似，它返回一个从 start 开始到 end 的 Buffer。但有点不一样的是，它并不是返回一个数据拷贝，而仅仅是返回一个包括这些索引数据的快照。所以一旦你对返回的结果进行处理时，原 Buffer 中的数据也会随着改变。

fieldBuf 在每一个迭代中已经从 0 重新索引，我们可以不用添加其他的索引计算，依照规范中的说明解析即可。我们看第一行的定义：

Byte	内 容	描 述
0-10	11 bytes	字段名

以下是解析出字段名称的代码:

```
var field = {};
```

```
field.name = fieldBuf.toString('ascii', 0, 11).replace(/\u0000/g, '');
```

❶

❶ 获取结果, 如: “LON” (longitude)。

默认情况下, `buf.toString()` 使用 UTF8 编码, Node Buffer 也支持其他编码格式,⁶包括例子中所使用的 ASCII。`buf.toString()` 同样支持你指定要解析的范围。如果字段数据长度比 11 位字节短, 则空位补 0, 我们得使用 `replace()` 来把空位的 0 替换为空字符, 避免以补 0 的字符 (`\u0000`) 结尾。

下一个有效的定义是字段的数据类型:

Byte	内 容	描 述
11	1 byte	字段类型

但是字符 C 和 N 对我们来说一点意义都没有, 深入探究规范之后, 我们可以从中获取更多的信息, 如表 3.3 所示:

表 3.3 字段类型

数据类型	数据输入
C (Character)	OEM 代码页字符
N (Numeric)	-.0123456789

这样很方便我们对这个数据进行处理。JavaScript 没有 `character` 和 `numeric`, 但是有 `String` 和 `Number`。在解析正式的记录时要记住这一点。现在用一个对象来记录这个映射关系:

```
var FIELD_TYPES = {
  C: 'Character',
  N: 'Numeric'
}
```

有了关系表后, 在解析二进制数据后, 可以直接获取得到有效的信息:

```
field.type = FIELD_TYPES[fieldBuf.toString('ascii', 11, 12)];
```

❶

⁶ 参见 http://nodejs.org/api/buffer.html#buffer_buffer。

❶ 获取结果，“Character”或“Numeric”。

`buf.toString()` 可以返回一个 ASCII 编码的字符，我们可以从映射哈希中获取完整的类型名称。

目前在字段描述中，剩下最后一个我们需要解析的信息，便是字段的长度。

Byte	内 容	描 述
16	1 byte	字段长度

下边是已经比较熟悉的代码：

```
field.length = fieldBuf[16];
```

❶ 获取结果：例如 435。

这一部分关于规范和每一个字段的描述处理代码的说明可以参考图 3.7。

数据记录

我们已经解析了头部信息，包括字段描述，我们还有一部分需要处理的，便是真正的数据记录。我们可以参考规范：

表文件的头部信息之后便是数据记录。数据记录前边是一个字节，如果记录没被删除，那么该字节为一个空格（20h）；如果记录被删除，那么该字节为一个星号（2Ah）。所有的字段都保存在记录中，没有字段分隔符或者记录终止符。文件以一个单字节的文件结束符结尾，OEM 代码页字符值为 26（1Ah）。

我们来逐行分析一下：

表文件的头部信息之后便是数据记录。

在这之前已经有 `fieldOffset` 来表示字节位置，头部有一个字段来表示头部信息的字节长度，我们存放在 `header.bytesInHeader`。可以从这个位置开始：

```
var startingRecordOffset = header.bytesInHeader;
```

我们已经从头部信息了解到很多东西。首先，可以确定有多少数据记录，`header.totalRecords` 已经记录了相关数据。其次，`header.bytesPerRecord` 有每一个记录分配了多少字节

的数据记录。明确了从哪个索引开始，需要多少迭代，每次迭代处理的数据量，我们可以构建一个很棒的迭代来处理每一条记录：

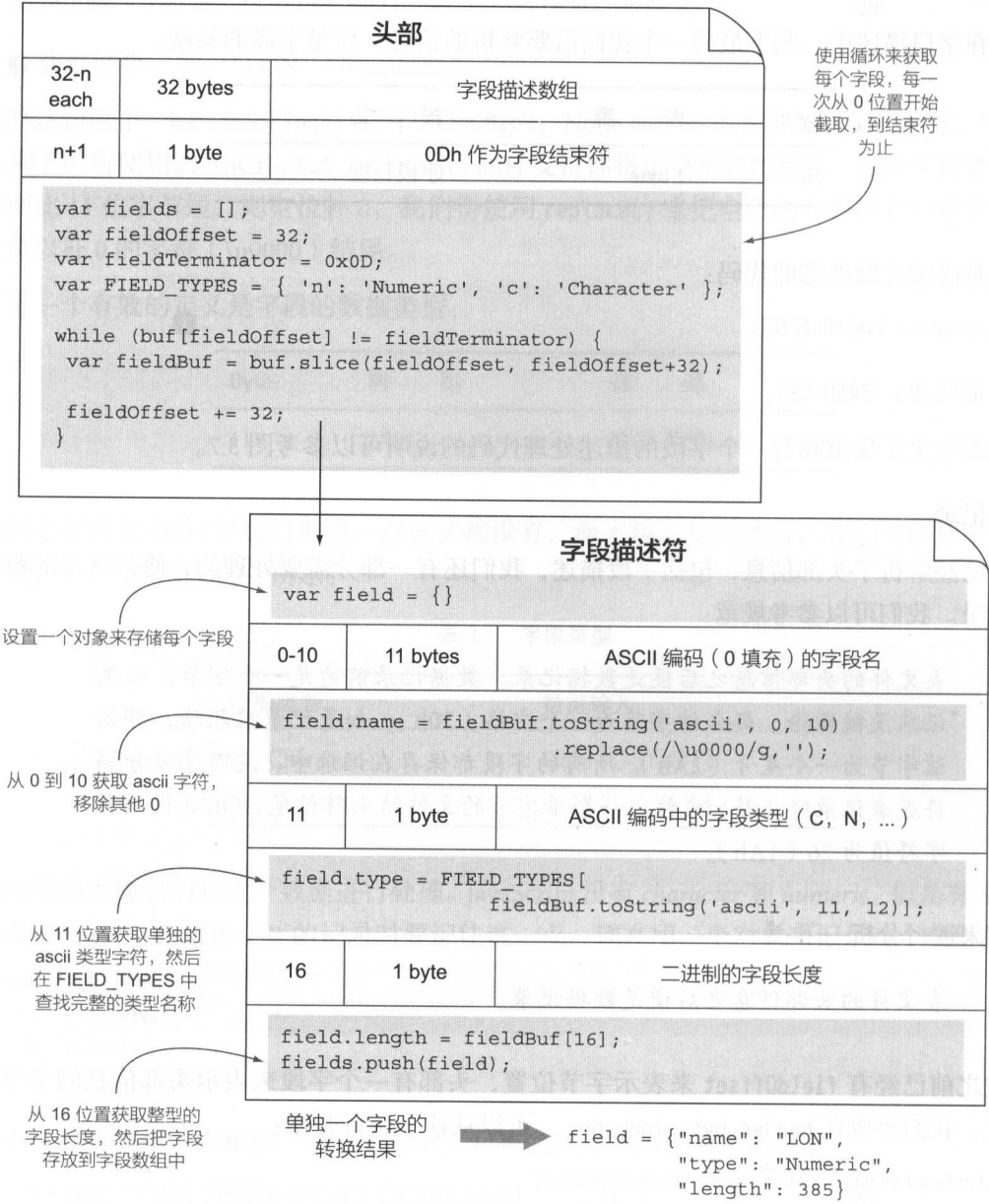


图 3.7 字段表述：使用 Node 的 Buffer API 将规范转换成代码


```
for (var i = 0; i < header.totalRecords; i++) {
    var recordOffset = startingRecordOffset + (i * header.bytesPerRecord);
    // 在这里解析每一条记录
}
```

现在，在每一个迭代的开始处，recordOffset 已经标示了我们要开始的索引位置。再来看一下规范：

数据记录前边是一个字节，如果记录没被删除，那么该字节为一个空格（20h），如果记录被删除，那么该字节为一个星号（2Ah）。

可以通过检查第一个字节来确定记录是否被删除：

```
var record = {};
record._isDel = buf.readUInt8(recordOffset) == 0x2A;
recordOffset++;
```

❶ 留意：我们也可以使用 buf[recordOffset]。

在与解析头部的时候测试是否为 fieldTerminator 一样，在这里，我们直接判断该整型字节是否匹配 0x2A 或者 ASCII 的星号字符。继续看一下规范：

所有的字段都保存在记录中，没有字段分隔符或者记录终止符。文件以一个单字节的文件结束符结尾，OEM 代码页字符值为 26（1Ah）。

最后，我们可以获取整个记录的数据。从字段描述数组的解析结果可以获取相关的信息，每一个字段中的 field.type、field.name、field.length（按字节数）都已经保存起来。我们希望可以以用户作为索引关键字来保存记录数据，并且把对应的值转化为对应的数据类型。简单的伪代码如下：

```
record[name] = cast type for (characters from length)
e.g.
record['pop2005'] = Number("13119679")
```

我们为每一个记录的字段都做对应的类型转化处理，需要另外一个 for 循环，如下：

```
for (var j = 0; j < fields.length; j++) {
    var field = fields[j];
    var Type = field.type == 'Numeric' ? Number : String;
    record[field.name] = Type(buf.toString('ascii', recordOffset,
        recordOffset+field.length).trim());
    recordOffset += field.length;
}
```

来看一下每个记录是如何处理的：

- 1. 首先，找出对应的 JavaScript 类型，保存为 Type。
- 2. 然后使用 buf.toString，从 recordOffset 开始，长度为 field.length，我们无法确定数据是否包括了无效的空格，所以使用 trim() 来清理掉。
- 3. 最后，给 recordOffset 加上 field.length 来调整索引，确保下次迭代可以从正确的索引开始。

这一部分关于规范和记录的解析处理代码的说明可以参考图 3.8。

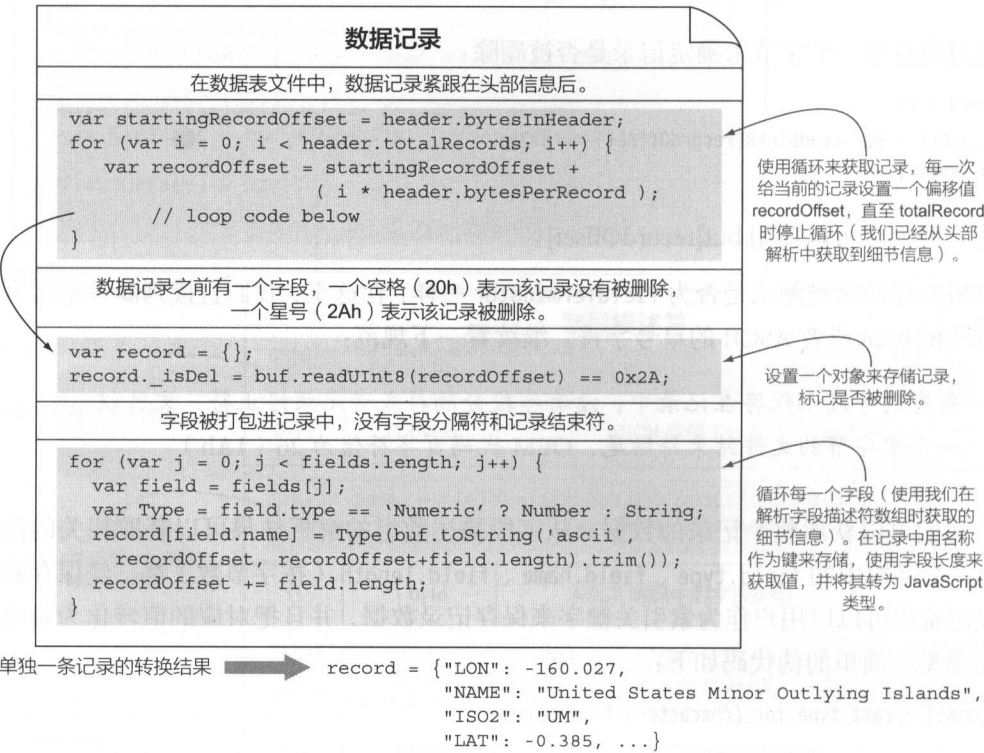


图 3.8 记录：使用 Node 的 Buffer API 将规范转换成代码

能跟上我吗？完整的代码如图 3.9 所示。

使用 Node Buffer API，可以把一个二进制文件转换为有用的 JSON 格式文件，整个应用运行后的输出大致如下：

```
{ header:
  { lastUpdated: 'Sat Aug 26 1995 21:55:03 GMT-0500 (CDT)',
```

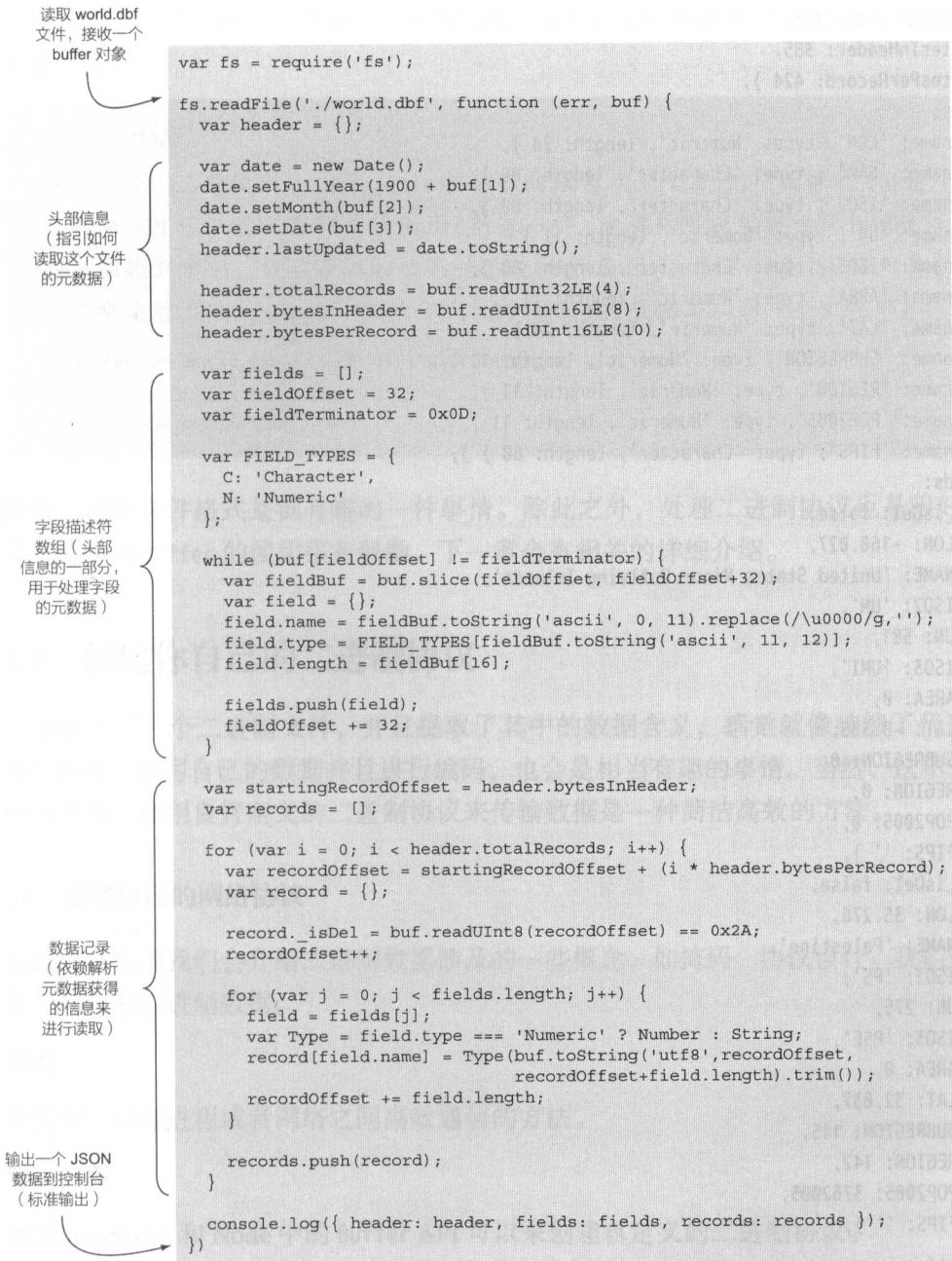


图 3.9 将 DBF 文件转成 JSON 的完整代码

```

    totalRecords: 246,
    bytesInHeader: 385,
    bytesPerRecord: 424 },
  fields:
  [ { name: 'LON', type: 'Numeric', length: 24 },
    { name: 'NAME', type: 'Character', length: 80 },
    { name: 'ISO2', type: 'Character', length: 80 },
    { name: 'UN', type: 'Numeric', length: 11 },
    { name: 'ISO3', type: 'Character', length: 80 },
    { name: 'AREA', type: 'Numeric', length: 11 },
    { name: 'LAT', type: 'Numeric', length: 24 },
    { name: 'SUBREGION', type: 'Numeric', length: 11 },
    { name: 'REGION', type: 'Numeric', length: 11 },
    { name: 'POP2005', type: 'Numeric', length: 11 },
    { name: 'FIPS', type: 'Character', length: 80 } ],
  records:
  [ { _isDel: false,
    LON: -160.027,
    NAME: 'United States Minor Outlying Islands',
    ISO2: 'UM',
    UN: 581,
    ISO3: 'UMI',
    AREA: 0,
    LAT: -0.385,
    SUBREGION: 0,
    REGION: 0,
    POP2005: 0,
    FIPS: '' },
    { _isDel: false,
    LON: 35.278,
    NAME: 'Palestine',
    ISO2: 'PS',
    UN: 275,
    ISO3: 'PSE',
    AREA: 0,
    LAT: 32.037,
    SUBREGION: 145,
    REGION: 142,
    POP2005: 3762005,
    FIPS: '' },
    ...
  ]
}
```

很神奇地，一个人类无法理解的二进制文件，可以转换成一种可读的、可以被有效地处理的，并且更有意义的格式。当然，这并不是什么黑魔法，只是需要时间去解读二进制

文件的规范，并且利用工具做一个解析转换。Buffer API 便提供了很棒的工具来处理二进制文件。

使用 fs 方法

我们也可以选择使用 `fs.writeFile` 把输出结果写入一个文件。^a就像在 Node 中的大部分 API 可以读取返回一个 Buffer 对象一样，大部分可以直接使用 Buffer 写入。在我们的例子中，最后获取的结果并不是一个 Buffer，而是一个 JSON 对象，所以需要使用 `JSON.stringify` 来结合 `fs.writeFile` 把数据写入到一个文件中：

```
fs.writeFile('world.json', JSON.stringify(result), ...
```

^a参考 <http://nodejs.org/api/fs.html>。

解析二进制文件格式是挺有趣的一件事情。除此之外，处理二进制协议也是很好玩的，并且对学习 Buffer 的使用很有帮助，下一章会有相关的详细介绍。

3.3 创建你自己的二进制协议

当你解析了一个二进制文件，并且提取了其中的数据含义，感觉就像破解了代码一样棒。同样，编写自己的数据并且进行编码，也会是相当有趣的事情。当然，这不仅仅是因为有趣。使用良好定义的二进制协议来传输数据是一种简洁高效的方法。

技巧 18 创建自己的网络协议

在这个技巧中我们会介绍二进制数据涉及的一些概念，如掩码、协议设计。我们还会涉及如何压缩二进制数据。

问题

你需要一种在进程或者网络之间高效通信的方法。

解决方案

使用 JavaScript 和 Node 中的 Buffer API 可以来创建自定义的二进制协议。

讨论

创建二进制协议，首先你需要确定传输哪些数据以及如何去表示这些数据。就像上一个技巧中一样，一个规范可以提供良好的实现思路。

在这个例子中，我们要开发一个简洁的数据库协议，主要内容有：

- 使用掩码来确定数据存放于哪个数据库。
- 数据保存以一个在 0~255 范围内的无符号正数（单字节）的键值来标识。
- 存储通过 zlib 压缩的任意长度的数据。

表 3.4 展示了我们如何写这个规范。

表 3.4 简单的键值数据库协议

Byte	内 容	描 述
0	1 byte	决定数据要写入到哪个数据库
1	1 byte	一个字节的无符号整数（0-255）用作数据库键存储
2-n	0-n bytes	存储的数据，任意通过 zlib 进行压缩的 byte

使用比特来表示选择哪个数据库

在我们定义的协议中，第一个字节用于表示选择哪个数据库来存储传输的数据。在数据接收端，主数据库就像一个简单的多维数组，支配着 8 个数据分库（恰好一个字节等于 8 比特）。在 JavaScript 中可以简单地使用数组字面量来表示：

```
var database = [ [], [], [], [], [], [], [], [] ];
```

比特对应位置的数据库将会存储接收到的数据。例如，数字 8 对应的二进制是 00001000。我们会把数据信息存储在第 4 个数据库中，因为第 4 个比特位是 1（比特按照从右往左的顺序）。

索引数据从 0 开始

在 JavaScript 中，数组索引是从 0 开始的，所以第 4 个数据库是在数组索引 3 的位置。为了减少误解，当提及比特在字节中位置时，我们使用 1 到 8 而非 0 到 7 来表示数据库的位置，这看起来会更加舒服。

如果你很好奇在 JavaScript 中，一个数字的二进制格式是怎么样的，可以使用内置的 toString 方法，传入参数 2 来查看：

```
8..toString(2) // '1000'
```

❶

❶ 调用一个数值的属性方法时，两个点（..）是必要的，第一个会被解析为小数点，而非属性操作符。

通常，数值的二进制不仅仅只有一位为 1。例如，20 的二进制表示是 00010100，在应用中则意味着我们要把数据存储在第 3 个和第 5 个数据库。

所以得想想办法，把给定的任意一个数值转换为只有一位为 1 的二进制。这里需要用到位掩码。位掩码可以达到我们想要的效果。例如，我们想要确定是否要把数据存放在第 5 个数据库，可以创建一个第 5 位为 1 的位掩码。用二进制的格式表示为 00010000，即是数值 32（或者十六进制的 0x20）。

之后使用这个掩码去测试某个数值是否满足要求，JavaScript 中有很多位操作符⁷可以利用，例如 &（位与）。& 操作符有点像 && 操作符，但不是判断两个条件都为真，而是判断两个比特位是否都为 1，如果是，则对应的位也为 1，举个例子：

```
000101000
& 000100000
-----
000100000
```

第 5 个比特位为 1，使用 & 操作符运算后依旧为 1。借由这个知识点，我们可以看到，如果一个值和对应的位掩码相同，或者本身为 1 时，通过运算后的值依旧还是对应的位掩码。有了这个信息，我们可以设置一个简单的条件来判断：

```
if ( (value & bitmask) === bitmask) { .. }
```

& 表达式需要用括号包裹起来，这是很重要的，否则，判断位掩码是否相等会先执行，这是操作符优先级导致的。⁸

为了测试接收到的二进制协议中的第一个字节，我们需要建立一个掩码列表，用来对应数据库的索引。如果匹配到对应的掩码，那么可以确定是哪个数据库需要写入数据。用一个数组来记录每一个二进制不同位为 1 的值：

```
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ]
```

这是一一对应的：

```
1      2      4      8      16     32     64     128
-----
00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000
```

现在我们知道如果一个字节在掩码数组中匹配到 1，那么它会匹配第 1 个数据库或者是数组中索引 0 的元素。我们可以设置一个简单的循环来测试每一个掩码对应的第一个字节的值：

⁷ 参见 https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/Bitwise_Operators。

⁸ 参见 https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/Operator_Precedence。


```
var database = [ [], [], [], [], [], [], [], [] ];
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ];

function store (buf) {
  var db = buf[0];
  bitmasks.forEach(function (bitmask, index) {
    if ( (db & bitmask) === bitmask) {
      // 当匹配到database[index]时
    }
  });
}
```

❶ 从 0 位置获取第一个字节。

刚开始接触位运算时会感觉非常棘手，但一旦你深入了解了它是如何工作的，则会变得十分高效。前边提到的相关内容不仅仅适用于 Node，在浏览器端的 JavaScript 也是可用的。我们的代码已经可以确定接受的数据存放在哪个数据库中，接下来需要确定数据要存储在哪个键值。

找出数据存储的键值

这是我们例子中最简单的部分了，在之前的技巧中已经学过。在开始前，我们先看一下规范中相关部分的说明，参考表 3.4。

Byte	内 容	描 述
1	1 byte	一个字节的无符号整数（0-255）用作数据库键存储

我们可以确定接收的数据中字节位为 1 的是一个无符号正数（0~255），用于表示数据库中存储数据的键值。我们特意把数据库设置为一个多维数据，第一维用于表示数据分库，第二维用来存储键值和数据，由于键值为数字，使用数组便可以了。⁹为了看起来更加直观，举个例子，数据 'foo' 存放在第一个和第三个数据库中的 0 键值，看起来是这样子的：

```
[
  ['foo'],
  [],
  ['foo'],
  [],
```

⁹尽管在即将到来的 ECMAScript 6 中会有更加理想的选择。


```
[],
[],
[],
[]
]
```

从字节位 1 中获取键值，可以使用我们熟悉的 `readUInt8` 方法：

```
var key = buf.readUInt8(1);
```

❶ 注意，使用 `buf[1]` 也可以。

我们把上述的代码加入之前的代码中：

```
var database = [ [], [], [], [], [], [], [], [] ];
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ];
```

```
function store (buf) {
  var db = buf[0];
  var key = buf.readUInt8(1);

  bitmasks.forEach(function (bitmask, index) {
    if ( (db & bitmask) === bitmask) {
      database[index][key] = 'some data';
    }
  });
}
```

❶ 'some data' 现在只是一个占位符，不代表真实数据。

现在我们把数据库索引和键值都解析出来了，接下来就是把要存储的数据搞定。

使用 zlib 解压缩

在传输字符串 / ASCII / UTF-8 数据时进行压缩绝对是个好主意，这可以大大减少传输使用的带宽。在简洁的数据库协议中，假定接收到的数据是已经压缩过的，我们稍后可以看看规范的相关内容。

在 Node 中内置的 `zlib` 模块提供了 `deflate`（压缩）、`inflate`（解压缩）的方法。它 also 包括了 `gzip` 的压缩方法。为了防止获取到错误的信息，需要检查接收到的数据是否经过了正确的压缩，如果不是，则不进行解压。通常，`zlib` 压缩的数据结果第一个字节为 `0x78`，¹⁰我们根据这个来进行判断：

¹⁰一个更加健壮的实现应该做更多的检查；参考 <http://tools.ietf.org/html/rfc6713>。

```
if (buf[2] === 0x78) { .. }
```

❶

❶ 请注意，我们从第 2 个字节位开始，因为前面是已经处理过的数据库索引和键值。

确定处理的是压缩过的数据后，可以使用 `zlib.inflate` 方法来解压缩。我们还需要使用 `buf.slice()` 来截取数据那一部分（保留前两个字节的话会报错）：

```
var zlib = require('zlib');
...
if (buf[2] === 0x78) {
  zlib.inflate(buf.slice(2), function (er, inflatedBuf) {
    if (er) return console.error(er);

    var data = inflatedBuf.toString();
  })
}
```

❶

❷

❶ 尽管我们做了检查，但是还是可能出现其他的错误情况，需要将错误打印出来，并且停止程序。

❷ `zlib.inflate` 返回一个 `Buffer` 对象，我们将其转换为 UTF-8 编码格式的字符串来存储。

万事俱备，我们可以使用自定义的简单数据库协议来存储数据了。让我们把所有的组件放在一起看一下：

```
var zlib = require('zlib');
var database = [ [], [], [], [], [], [], [], [] ];
var bitmasks = [ 1, 2, 4, 8, 16, 32, 64, 128 ];
```

```
function store (buf) {
  var db = buf[0];
  var key = buf.readUInt8(1);

  if (buf[2] === 0x78) {
    zlib.inflate(buf.slice(2), function (er, inflatedBuf) {
      if (er) return console.error(er);

      var data = inflatedBuf.toString();

      bitmasks.forEach(function (bitmask, index) {
        if ( (db & bitmask) === bitmask) {
          database[index][key] = data;
        }
      });
    });
  }
}
```

❶

❶ 数据存放在对应的数据库以及对应的键值中。

现在我们的代码可以用来存储数据了。但是还需要生成数据，代码如下：

```
var zlib = require('zlib');
var header = new Buffer(2);

header[0] = 8;
header[1] = 0;

zlib.deflate('my message', function (er, deflateBuf) {
  if (er) return console.error(er);
  var message = Buffer.concat([header, deflateBuf]);
  store(message);
})
```

❶
❷
❸
❹
❺

❶ 存放在第 4 个数据库（8 = 00001000）。

❷ 存放在 0 键值。

❸ 压缩 'my message'。

❹ 把头部信息和数据打包在一起。

❺ 存储信息。

我们还可以编写一个在 TCP 中发送信息并且处理错误的例子，但是暂时歇一下，等你学习了下一章的 Node 网络后，把它当成练习来尝试解决。

3.4 总结

在这一章中学习了 Buffers 以及如何使用 toString 方法把 Buffers 转换为其他编码格式的字符串。我们深入了解了如何使用 Buffer API 来把二进制文件转换为可读的格式。最后，我们学习掩码和压缩，知道了如何去创建自己的通信协议。

这一章深入浅出地介绍了 Buffers 在 Node 中常见的使用方式，希望能够让读者们更加熟悉地使用 Buffers。现在你大可以尝试编写一个二进制转换工具并且发到 npm 上，或者一个更好满足你业务需求的协议正等着你来创造。

在下一章，我们将会看到另一个 Node 核心部分——事件。

4

Events: 玩转 EventEmitter

本章概要

- 使用 Node 的 EventEmitter 模块
- 异常管理
- 第三方模块中如何使用 EventEmitter
- 如何使用 domains 模块的 events
- EventEmitter 的替代品

Node 的事件模块目前只包含一个类：EventEmitter。这个类在 Node 的内置模块以及第三方模块中被大量使用。许多 Node 项目的架构都是用它实现的。因此理解 EventEmitter 以及掌握如何使用它是非常重要的。

它是一个简单的类，假如你熟悉 DOM 或者是 jQuery 的 events，那你对它一定不会陌生。在使用 Node 中最需要考虑的是异常处理，我们在技巧 21 中会详细讲解。

EventEmitter 可以用在很多地方，它通常被用作基类来解决大量的问题，从创建服务器到构建业务逻辑。实际上，它被用来作为像 Express 那样受欢迎的 Node 模块的核心类的基类，学习它是如何工作的将对写出地道的 Node 代码有帮助并且对现有的模块也是相得益彰。

在这一章中，你不仅将学习如何使用 `EventEmitter` 来创建类，并且在 `Node` 核心模块以及开源模块中是如何使用它的。而且你将学习如何解决在使用 `EventEmitter` 时碰到的问题和一些替代方法。

4.1 基础用法

要使用 `EventEmitter`，首先必须先继承于它。这一章节涵盖的技术包括从 `EventEmitter` 继承以及把它融入一些已经从其他基类继承的类。

技巧 19 从 `EventEmitter` 继承

这个技术展示了如何创建一个基于 `EventEmitter` 的自定义类。在理解这个技术的原理后，你将学习到如何使用 `EventEmitter`，以及如何更好地使用基于它的模块。

问题

你希望通过事件驱动的手段来解决问题。你有一个类你希望在异步事件发生的时候来操作它。

不管是网页应用，桌面应用还是手机应用的用户界面都有一个相同的特点：它们都是事件驱动的。事件是解决那些本质是异步问题的最好的典范：来自人类的输入。我们将用一个音乐播放器作为例子来展示 `EventEmitter` 是如何工作的。它不会真正播放音乐，但是它的基本概念是理解如何使用事件的好方法。

解决方案

在 `Node` 中使用事件的典型例子是从 `EventEmitter` 继承。可以通过一个简单的原型类实现，只要记得在你的类的构造函数中调用 `EventEmitter` 的构造函数。

第一个清单展示了如何从 `EventEmitter` 继承。

例子 4.1 从 `EventEmitter` 继承

```
var util = require('util');  
var events = require('events');
```

```
function MusicPlayer() {  
  events.EventEmitter.call(this);  
}
```

```
util.inherits(MusicPlayer, events.EventEmitter);
```

1

1 使用 `util.inherits` 从原型类继承在 `Node` 中是地道的做法。

讨论

结合一个简单的构造函数和 `util.inherits` 是创建自定义事件驱动类的最简单也是最常见方法。下一个代码清单扩展了前一个例子，并且展示了如何通过 `emit` 来触发事件，以及如何通过 `on` 来绑定监听器。

例子 4.2 从 EventEmitter 继承

```
var util = require('util');
var events = require('events');
var AudioDevice = {
  play: function(track) {
    // Stub: Trigger playback through iTunes, mpg123, etc.
  },

  stop: function() {
  }
};

function MusicPlayer() {
  this.playing = false;
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);

var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.playing = true;
  AudioDevice.play(track);
});

musicPlayer.on('stop', function() {
  this.playing = false;
  AudioDevice.stop();
});

musicPlayer.emit('play', 'The Roots - The Fire');

setTimeout(function() {
  musicPlayer.emit('stop');
}, 1000);
```

❶ 可以配置类的状态，稍后在 `EventEmitter` 的构造器会按需被调用到。

❷ 这里的 `inherits` 方法将方法从一个原型拷贝到另外一个原型，这是基于 `EventEmitter` 创建类的通用模式。

❸ `emit` 方法用于触发事件。

这看上去好像没什么，但假如我们在 `play` 触发时需要做些别的，比如用户界面需要更新。对 `play` 事件添加一个新的监听器就能轻松实现。下面的代码片段展示了如何添加更多的监听器。

例子 4.3 添加多个监听器

```
var util = require('util');
var events = require('events');

function MusicPlayer() {
  this.playing = false;
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);

var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.playing = true;
});

musicPlayer.on('stop', function() {
  this.playing = false;
});

musicPlayer.on('play', function(track) {
  console.log('Track now playing:', track);
});

musicPlayer.emit('play', 'The Roots - The Fire');

setTimeout(function() {
  musicPlayer.emit('stop');
}, 1000);
```

❶

❶ 可以按需添加新的监听器。

监听器也能被删除，`emitter.removeListener` 可以将一个监听器从一个指定的事件上删除。`emitter.removeAllListeners` 可以删除全部的监听器。你需要将一个监听器保存在一

个变量中，使它在被删除时能被引用到，就像通过 `clearTimeout` 来删除 `timers` 一样。下一个例子展示了如何删除监听器。

例子 4.4 删除监听器

```
function play(track) {  
  this.playing = true;  
}  
  
musicPlayer.on('play', play);  
  
musicPlayer.removeListener('play', play);
```

❶ 如果想要移除一个监听，那么一个监听的引用是不可或缺的。

`util.inherits` 通过封装 ES5 的 `Object.create` 方法来实现，它通过从一个原型到另外一个原型继承属性的方式来实现。Node 中还将父类的构造函数保存在 `super_` 属性中。这使得调用父类构造函数变得简单许多。在使用 `util.inherits` 之后，你的原型类可以通过 `YourClass.super_` 调用 `EventEmitter`。

你也可以仅对一个事件反应一次，而不是每次它触发时都反应。通过 `once` 方法来绑定监听器就能实现。这在一个事件虽然会触发多次，但你只关心它第一次触发的时候非常有用。例如，你可以通过更新例子 4.3 来追踪播放器的播放事件是否已经触发了。

```
musicPlayer.once('play', {  
  this.audioFirstStarted = new Date();  
});
```

当从 `EventEmitter` 继承时，记得要在你的构造函数中通过 `events.EventEmitter.call(this)` 来调用 `EventEmitter` 的构造函数。因为这样会将这个实例附加到一个已经启用的 `domain` 中，我们会在技巧 22 中学到更多关于 `domain` 的知识。

我们现在所学到的这几个方法——`on`、`emit`，以及 `removeListener`，在 Node 的开发中是基本的知识。一旦你掌握了 `EventEmitter`，你会发现它到处都会被用到，无论是在 Node 的模块中还是在其他的地方，通过 `net.createServer` 创建的 `tcp/ip` 服务将会返回一个基于 `EventEmitter` 的对象，甚至 `process` 对象也是一个 `EventEmitter` 实例。另外，主流的框架 `Express` 也是基于 `EventEmitter` 的，你可以创建一个 `express` 对象 `app`，并且调用 `app.emit`，将消息在整个系统中传递。

技巧 20 混合 EventEmitter

有些时候集成 `EventEmitter` 并不是最好的方式，这个时候可以通过混合 `EventEmitter`。

问题

这是技巧 19 的替代方案，除了将 EventEmitter 作为父类来继承，还可以将它的方法拷贝到另外一个类中。这种情况适用于当你有一个现成的类，并且不能简单地将它继承 EventEmitter 的时候很有用。

解决方案

通过 for-in 循环就足以将属性从一个原型对象拷贝到另一个原型对象上。这样，你可以只拷贝你需要的那些属性。

讨论

这个例子看似简单，然而有时候相比继承，拷贝 EventEmitter 的属性来得更实用，这个方法更像多继承，请看下面的例子。

例子 4.5 混合 EventEmitter

```
var EventEmitter = require('events').EventEmitter;

function MusicPlayer(track) {
  this.track = track;
  this.playing = false;

  for (var methodName in EventEmitter.prototype) {
    this[methodName] = EventEmitter.prototype[methodName];
  }
}

MusicPlayer.prototype = {
  toString: function() {
    if (this.playing) {
      return 'Now playing: ' + this.track;
    } else {
      return 'Stopped';
    }
  }
};

var musicPlayer = new MusicPlayer('Girl Talk - Still Here');

musicPlayer.on('play', function() {
  this.playing = true;
  console.log(this.toString());
});
```

```
musicPlayer.emit('play');
```

❶ 这个是为了拷贝相关属性的 for-in 循环。

一个多继承的例子是 Connect 框架。¹核心的 Server 类从多个类继承，Connect 的作者决定自己实现属性拷贝的方法，看下面的例子。

例子 4.6 Connect 中的 util.merge 方法

```
exports.merge = function(a, b){
  if (a && b) {
    for (var key in b) {
      a[key] = b[key];
    }
  }
  return a;
};
```

当你已经有一个很强大的类，而且这个类想利用 events 对象而不去继承它的时候，这个技巧就很有用。

一旦你继承了 EventEmitter，你就要处理异常，下一节我们将会看到怎样处理 EventEmitter 产生的异常。

4.2 异常处理

尽管大多数的事件处理起来差不多，但 error 事件比较特殊，所以需要特殊对待。这一节会看到两种处理异常的方法，一个是通过添加对 error 事件的监听器，还有一个是通过 domain 来收集在这个 domain 之中的 EventEmitter 实例产生的异常。

技巧 21 管理异常

通过 EventEmitter 来处理异常需要遵守它的游戏规则。这个技巧展示了异常处理是如何工作的。

问题

你正在使用 EventEmitter，并且在异常发生时希望能够优雅地处理，但它却不断地抛出异常。

¹ 参见 <http://www.senchalabs.org/connect/>。

解决方案

要在异常发生时阻止异常抛出，只要在 `error` 事件上添加一个监听器，任何从 `EventEmitter` 继承的自定义类，或者标准类都可以通过这个方法来解决。

讨论

要处理异常，对 `error` 事件绑定一个监听器。下面的例子通过创建一个音乐播放器来向你展示这个技巧。

例子 4.7 基于事件的错误

```
var util = require('util');
var events = require('events');

function MusicPlayer() {
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);

var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.emit('error', 'unable to play!');
});

musicPlayer.on('error', function(err) {
  console.error('Error:', err);
});

setTimeout(function() {
  musicPlayer.emit('play', 'Little Comets - Jennifer');
}, 1000);
```

❶ 监听 `error` 事件。

这个例子可能看上去简单，但它很有用，它能帮助你理解 `EventEmitter` 如何处理异常。它感觉像是一个特殊的例子，也确实是。下面一段话是从 Node 的文档中摘录的引用。

当一个 `EventEmitter` 实例发生错误时，通常会发出一个 `error` 事件。在 Node 中，`error` 事件被当作特殊的情况，假如没有监听器，那么默认的动作是打印一个堆栈并退出程序。

你可以试下在例子 4.7 中把 'error' 的 handler 删除。应该在终端中显示一个堆栈信息。这样做是合理的，否则在缺乏错误处理的时候会导致意想不到的结果。事件的名字或者类型必须和 error 看上去完全一样，多余的空格、符号或者大写字母都不被认为是 error 事件。

这个规定意味着基于事件的异常处理有着很大的一致性。这可能是一个特例，但值得关注。

技巧 22 通过 domains 管理异常

处理多个 EventEmitter 实例的异常感觉非常困难，除非通过 domains。

问题

你正在处理多个非阻塞的 API，但纠结于如何有效地处理异常。

解决方案

Node 的 domain 模块能被用来集中地处理多个异步操作，这包括 EventEmitter 实例发出的未处理的 error 事件。

讨论

Node 的 domain 接口提供了用异常处理封装已有的非阻塞 API 以及错误的方法。这能帮助集中处理异常，而且在多个互相依赖的 I/O 操作时非常有用。

例子 4.8 通过用两个 EventEmitter 来构建一个音乐播放器程序，通过这个例子来展示如何用异常处理器来处理多个对象。

例子 4.8 通过 domain 管理异常

```
var util = require('util');
var domain = require('domain');
var events = require('events');
var audioDomain = domain.create();

function AudioDevice() {
  events.EventEmitter.call(this);
  this.on('play', this.play.bind(this));
}

util.inherits(AudioDevice, events.EventEmitter);

AudioDevice.prototype.play = function() {
```

❶

```
this.emit('error', 'not implemented yet');
};

function MusicPlayer() {
  events.EventEmitter.call(this);

  this.audioDevice = new AudioDevice();
  this.on('play', this.play.bind(this));

  this.emit('error', 'No audio tracks are available');
}

util.inherits(MusicPlayer, events.EventEmitter);

MusicPlayer.prototype.play = function() {
  this.audioDevice.emit('play');
  console.log('Now playing');
};

audioDomain.on('error', function(err) {
  console.log('audioDomain error:', err);
});

audioDomain.run(function() {
  var musicPlayer = new MusicPlayer();
  musicPlayer.play();
});
```

❶ Domain 模块必须被加载，然后方可利用 create 方法创建一个相应的实例。

❷ 这个错误以及任何其他错误都会被同一个 error 处理方法所处理。

❸ 任何在这个回调中导致错误的代码都会被 domain 覆盖到。

Domain 可以与 EventEmitter 的子类、网络代码，并且还有异步文件系统的方法一起使用。要可视化域名是如何工作的，想象 domain.run 回调方法包装你的代码，即使在回调中的代码会触发事件发生在它的外面。被抛出的任何错误仍然会被捕获。图 4.1 说明了这一过程。

没有 domain，使用 throw 抛出的异常可能将程序处于未知状态。domain 避免这种情况，并帮助你更优雅地处理异常。

现在你知道如何继承 EventEmitter 处理错误，你应该开始看到各种有用的方法。下一节通过引入一些先进的使用模式和更高级别的与事件相关的编程结构问题的解决方案，以此来拓宽这些技术。

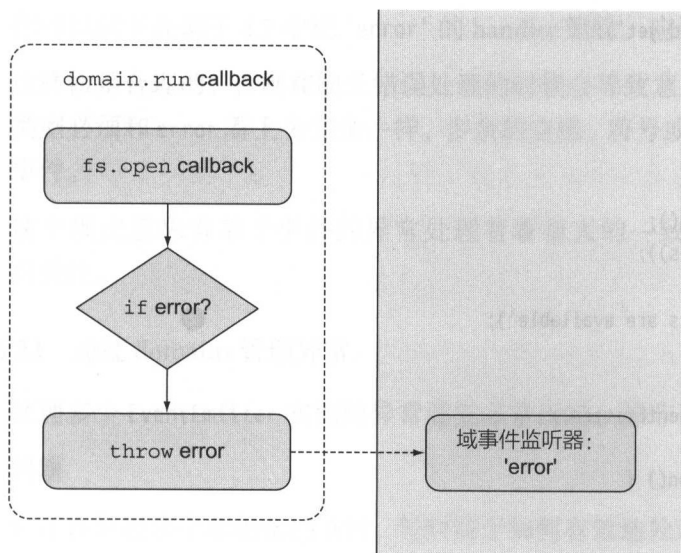


图 4.1 Domain 帮助捕捉异常和使用 EventEmitter 风格的 API 处理异常

4.3 高级模式

本节提供了一些最佳实践技术来解决在使用 EventEmitter 时发现的结构性问题。

技巧 23 反射

有些时候你需要动态地响应一个 EventEmitter 实例的变化，或者查询它的监听器。这个技巧将解释如何做到这一点。

问题

你需要知道一个监听器何时被添加到一个 emitter 上，或者查询现有的监听器。

解决方案

要追踪监听器何时被添加，EventEmitter 发出一个特殊的事件叫 `newListener`。监听了这个事件的监听器会接受到事件的名字以及监听器的方法。

讨论

在某些方面，写出好的 Node 代码和伟大的 Node 代码的区别就在于对 EventEmitter 的理解。能够正确反映 EventEmitter 对象，能够创建更灵活和直观的 API。一个动态的方

法是通过 `newListener` 事件，这个事件会在监听器通过 `on` 方法添加的时候触发。有趣的是，这个事件本身是通过 `EventEmitter` 自己触发、通过 `emit` 实现的。

下面的例子展示了如何追踪 `newListener` 事件。

例子 4.9 密切关注新的监听

```
var util = require('util');
var events = require('events');

function EventTracker() {
  events.EventEmitter.call(this);
}

util.inherits(EventTracker, events.EventEmitter);

var eventTracker = new EventTracker();

eventTracker.on('newListener', function(name, listener) {
  console.log('Event name added:', name);
});

eventTracker.on('a listener', function() {
  // This will cause 'newListener' to fire
});
```

❶ 一旦有监听器加入进来就开始追踪。

尽管 `'a listener'` 从未显式地在这个例子中触发事件，但是 `newListener` 事件仍然会触发。因为监听器的回调函数传递以及事件名称，是一个很好的方式来为需要访问原始监听的功能创建简单的 API，例子 4.10 演示了这个概念，自动启动一个定时器，来添加监听器脉冲事件。

例子 4.10 在有新建监听器事件中自动触发事件

```
var util = require('util');
var events = require('events');

function Pulsar(speed, times) {
  events.EventEmitter.call(this);

  var self = this;
  this.speed = speed;
  this.times = times;
```

```

    this.on('newListener', function(eventName, listener) {
      if (eventName === 'pulse') {
        self.start();
      }
    });
  }

util.inherits(Pulsar, events.EventEmitter);

Pulsar.prototype.start = function() {
  var self = this;
  var id = setInterval(function() {
    self.emit('pulse');
    self.times--;
    if (self.times === 0) {
      clearInterval(id);
    }
  }, this.speed);
};

var pulsar = new Pulsar(500, 5);

pulsar.on('pulse', function() {
  console.log('.');
});

```

❶ 每一次暂停都显示一个点。

我们可以更进一步，并通过 `emitter.listeners(event)` 查询 `EventEmitter` 对象上的事件监听。虽然所有监听事件不能一次性返回，但整个列表是可以通过 `this._events` 对象获取的，尽管这个属性应视为私有。`listeners` 方法目前返回一个数组实例。这可以被用来遍历多个 `Listener`，如果有多个 `listener` 被添加到指定的事件，可能会在异步进程结束后删除它们，或者检查是否有任何的 `Listener` 被添加。

当一个事件的数组可以被获取到的情况下，`listeners` 函数将返回 `this._events[type].slice(0)`。对一个数组调用 `slice` 是用来创建一个数组的拷贝。文档上说，这个行为在将来可能会改变，所以假如你真想得到已添加的 `listener` 的拷贝，那你可以自己调用 `slice` 方法来确保得到的是拷贝而不是实际的引用。

例子 4.11 给 `Pulsar` 类添加了一个 `stop` 方法。当 `stop` 方法被调用时，它会检查，看看是否有任何的 `listener`，否则，它将引发错误。检查监听器是一个很好的方式来防止不正确的用法，但这并不是必需的。

例子 4.11 查询监听器

```
Pulsar.prototype.stop = function() {  
  if (this.listeners('pulse').length === 0) {  
    throw new Error('No listeners have been added!');  
  }  
};  
  
var pulsar = new Pulsar(500, 5);  
  
pulsar.stop();
```

技巧 24 探索 EventEmitter

许多成功的开源 Node 模块都是基于 EventEmitter 的,知道在什么地方使用 EventEmitter,并知道如何利用它是非常有用的。

问题

假如你正在开发一个大项目,有很多模块之间需要互相通信。

解决方案

无论你在使用 Node 的标准模块或者开源类库,找一下 emit 以及 on 这两个方法。比如,Express 模块中的 app 对象就有这些方法,并且它们非常适合在应用中发送消息。

讨论

通常,当你操作一个大项目时,你会有一个主要的核心模块。如果你正在使用 Express 构建一个 web 应用,那么 app 对象就是这样的一个模块。简单地查看源码就会发现该对象继承于 EventEmitter,这样就可以利用事件来使不同组件通信。

例子 4.12 显示,其中一个监听器绑定到一个事件,当一个特定的路由被访问时,这个事件将被触发。

例子 4.12 在 Express 中复用 EventEmitter

```
var express = require('express');  
var app = express();  
  
app.on('hello-alert', function() {  
  console.warn('Warning!');  
});
```

```
app.get('/', function(req, res){
  res.app.emit('hello-alert');
  res.send('hello world');
});

app.listen(3000);
```

❶ app 对象在 res.app 也是可得的。

虽然这看上去有些故弄玄虚，然而如果 router 被定义在另外一个文件中，那么你就无法访问 app 对象，除非它被定义成全局对象。

另外一个著名的基于 EventEmitter 的项目是 Redis 客户端 (<https://npmjs.org/package/redis>)。RedisClient 的实例继承于 EventEmitter。这使你可以挂接到有用的事件，如错误事件。

例子 4.13 在 redis 模块中复用 EventEmitter

```
var redis = require('redis'),
    var client = redis.createClient();

client.on('error', function(err) {
  console.error('Error:', err);
});

client.on('monitor', function(timestamp, args) {
  console.log('Time:', timestamp, 'arguments:', args);
});

client.on('ready', function() {
  // Start app here
});
```

❶ 这个 monitor 的事件是通过 redis 模块颁布的，用于追踪内部是否有活动发生。

当使用拆分路由技术将路由放在不同的文件中时，你可以通过调用 res.app.emit(event) 来发送事件。

这似乎是一个特定的 Express 例子，但其他流行的开源模块也是基于 EventEmitter 的，找一下 emit 以及 on 这两个方法。Node 的内部模块像 process 对象与 net.createServer，也继承于 EventEmitter，并且一些著名的开源项目也是继承于这些模块的。这意味着有大量的基于事件的解决方案架构的问题。

这个例子也突出了使用 EventEmitter 的另一个好处，异步进程可以尽快做出反应。如果 hello-alert 事件进行一个非常缓慢的操作，如发送电子邮件，浏览页面的用户可能不想

等待这个过程完成。在这种情况下，可以呈现所请求的页面，同时有效地在后台执行较慢的操作。

Node Redis 的客户端非常优秀地使用 `EventEmitter`，作者对每个方法都写了详细的文档。这是一个好主意，如果有人加入你的项目，他们可能会发现很难知道总共有多少事件正在被使用。

技巧 25 组织事件名称

有些项目中存在着太多的事件。这个技巧展示了如何处理由于写错事件名造成的缺陷。

问题

你已经对你项目中的事件失去控制，并且担心太容易写错一个事件名造成一个难以检查的 bug。

解决方案

最简单的方法是用一个对象来存放所有的事件名。这样在项目中创建了一个统一存放事件的地方。

讨论

很难跟踪整个项目凌乱的事件名。管理它的一个方法是将每个事件名称保持放在一个地方。例子 4.14 在本章上一个例子的基础上演示了使用对象为事件名称分类。

例子 4.14 使用对象组织管理事件名称

```
var util = require('util');
var events = require('events');

function MusicPlayer() {
  events.EventEmitter.call(this);
  this.on(MusicPlayer.events.play, this.play.bind(this));
}

var e = MusicPlayer.events = {
  play: 'play',
  pause: 'pause',
  stop: 'stop',
  ff: 'ff',
  rw: 'rw',
  addTrack: 'add-track'
};
```

```
util.inherits(MusicPlayer, events.EventEmitter);

MusicPlayer.prototype.play = function() {
  this.playing = true;
};

var musicPlayer = new MusicPlayer();

musicPlayer.on(e.play, function() {
  console.log('Now playing');
});

musicPlayer.emit(e.play);
```

- ❶ 用一个对象存储事件名称，可以像这样罗列清晰。
- ❷ 当添加新的事件的时候，类的用户可以引用对象中的事件的列表而不是用字符串硬编码事件名称。

虽然 `EventEmitter` 是 Node 的标准库的一个部分，可以优雅地解决很多问题，但它会导致很多大型项目的缺陷，人们可能会忘记某一特定事件的名称的来源。解决这个问题的一种方法是，避免用字符串来作为事件名。相反，一个对象可以被用于引用该事件名称的字符串的属性。

如果你正在写一个可重复使用的、开放源码的模块，你应该考虑这部分的公共 API，因此很容易让人们得到事件名称的列表集。

还有其他观察者模式的实现，避免使用字符串事件名称、有效类型检查的事件。在下一个技巧中，我们来看一些可以通过 NPM 的模块。

虽然 `EventEmitter` 在 Node 项目时提供了很多解决方案，但还有替代方案。接下来的一节介绍一些主流的替代方案。

4.4 第三方模块以及扩展

`EventEmitter` 本质上是一个观察者模式的实现。这种模式是一种可以帮助扩展 Node 在多个进程或者网络中运行的模式。接下来的技巧介绍了一些由 Node 社区创建更受欢迎的替代方案。

技巧 26 EventEmitter 的替代方案

EventEmitter 有非常好的 API，但有时一个问题需要一个不同的解决方案。让我们来探索一些其他方案来替代 EventEmitter。

问题

你正在尝试解决一个 EventEmitter 不能很好处理的问题。

解决方案

这取决于你试图解决的问题的确切性质，有几种 EventEmitter 的替代方案：发布/订阅、AMQP 和 js-signals，是 Node 一些很好支持的流行的替代品。

讨论

该 EventEmitter 类是观察者模式的实现。一个类似的模式是发布/订阅，发布者在这里发送消息而不需要知道订阅者具体是如何实现的。

发布/订阅模式在需要水平扩展的情况下非常有用。如果你需要在多个服务器上运行多个 Node 进程，像 AMQP 和 ØMQ 这样的技术可以帮助实现这一点。它们都专门设计来解决这一类问题，假如你已经使用 Redis，那可能没有像 Redis 的 publish/subscribe 那样方便。

如果你需要水平扩展一个分布式的集群，那么一个 AMQP 比如 RabbitMQ 将非常好用。rabbitmq-nodejs-client 模块有一个 publish/subscribe 的 API。下面的例子显示了一个使用 RabbitMQ 的简单例子。

例子 4.15 在 Node 中使用 RabbitMQ

```
var rabbitHub = require('rabbitmq-nodejs-client');
var subHub = rabbitHub.create( { task: 'sub', channel: 'myChannel' } );
var pubHub = rabbitHub.create( { task: 'pub', channel: 'myChannel' } );

subHub.on('connection', function(hub) {
  hub.on('message', function(msg) {
    console.log(msg);
  }).bind(this));
});
subHub.connect();

pubHub.on('connection', function(hub) {
  hub.send('Hello World!');
});
pubHub.connect();
```

❶ 当收到消息时候打印消息。

ØMQ (<http://www.zeromq.org/>) 在 Node 社区中更受欢迎。Justin Tulloss 和 TJ Holowaychuk 的 `zeromq.node` 模块 (<https://github.com/JustinTulloss/zeromq.node>) 是一个比较受欢迎的实现。下面的例子显示这个 API 是多么简单。

例子 4.16 使用 ØMQ

```
var zmq = require('zmq');
var push = zmq.socket('push');
var pull = zmq.socket('pull');

push.bindSync('tcp://127.0.0.1:3000');
pull.connect('tcp://127.0.0.1:3000');
console.log('Producer bound to port 3000');

setInterval(function() {
  console.log('sending work');
  push.send('some work');
}, 500);

pull.on('message', function(msg) {
  console.log('work: %s', msg.toString());
});
```

假如你已经在 node 中使用 Redis 了，那么值得尝试一下它的 pub/sub API。例子 4.17 展示了使用 Node redis 客户端的例子。

例子 4.17 使用 Redis

```
var redis = require('redis');
var client1 = redis.createClient();
var client2 = redis.createClient();
var msg_count = 0;

client1.on('subscribe', function(channel, count) {
  client2.publish(channel, 'Hello world.');
```

❶

```
});

client1.on('message', function(channel, message) {
  console.log('client1 channel ' + channel + ': ' + message);
  client1.unsubscribe();
  client1.end();
  client2.end();
});
```

```
client1.subscribe('channel');
```

❶ 当使用 Redis 模块时候，确保关闭客户端连接。

最后，如果 publish/subscribe 不是你要找的，那么你可能想看看 js-signal (<https://github.com/millermedeiros/js-signals>)。该模块是一个消息系统，不使用字符串作为信号名称、派遣或监听那些尚不存在的事件，会引发错误。

例子 4.18 显示了 js-signal 如何发送和接收消息。注意 signals 是一个对象的属性，而不是字符串，那些监听器可接收任意数量的参数。

例子 4.18 使用 js-signal

```
var signals = require('signals');
var myObject = {
  started: new signals.Signal()
};

function onStarted(param1, param2){
  console.log(param1, param2);
}

myObject.started.add(onStarted);
myObject.started.dispatch('hello', 'world');
```

❶

❷

❶ 在已启动的信号量上绑定一个监听。

❷ 使用两个参数分发信号。

正如在技巧 25 中提到的，js-signals 提供了一种使用属性作为信号名称的方法，但分发事件到没有注册的监听器会引发异常。这种方法更像强类型的事件，和其他大多数的 pub/sub 实现有着很大的不同。

4.5 总结

在这一章中，你已经学会了在继承和多继承中如何使用 EventEmitter，以及如何在不使用 domain 的情况下管理错误。你还看到了如何集中管理事件名称、模块开源如何使用 EventEmitter，以及一些替代解决方案。

你应该从本章学习到，虽然 EventEmitter 通常用作继承的基类，但它也可以混合现有类。此外，虽然 EventEmitter 在很多问题上是一个很好的解决方案并且在 Node 内部大量使用，有时还有其他更优的解决方案。例如，如果你使用 Redis，那么你可以利用它的发

布/订阅的实现。最后，EventEmitter 也不是没有问题的，管理大量事件名称会导致错误，不过现在你知道如何使用一个对象充当事件的名称来避免这种情况了。

在下一章中，我们将看到一个相关的话题：**stream**。stream 是建立在一个基于事件的 API，所以你也可以使用 EventEmitter 中的一些技巧。

5 流：最强大和最容易误解的功能

本章概要

- 什么是流和怎么使用
- 如何使用 Node 集成的流 API
- 流 API 在 Node 0.8 版本以下的使用
- 0.10 版本以后流的原始类
- 测试流的策略

流是基于事件的 API，用于管理和处理数据，而且有不错的效率。借助事件和非阻塞 I/O 库，流模块允许在其可用的时候动态处理，在其不需要的时候释放掉。

数据流并不是新东西，但是贯穿 Node 整体的概念。第 4 章以后，掌握流的使用才能真正胜任 Node 开发。

流的核心模块为构建基于事件的流类提供抽象工具。这就意味着你可以使用模块实现流，而不需要自己实现。但是为了最大限度地利用流，理解它的工作原理非常重要。本章设计思路：理解流，使用 Node 内建的流 API，最后创建和测试自己的流。尽管流模块的概念很抽象，一旦你掌握主要概念，你将会开始看到流在各种场合的用途。

下面介绍流的高级别概况，和 Node 0.10 支持的这两个 API。

5.1 流的介绍

在 Node 中，流是由几个不同的对象附着的抽象接口。谈及流，我们指的是在特定场景中，它们是一个协议。流是能够读写的，并且是基于事件实现的一个实例，更多事件相关内容见第 4 章。流提供创建对象间数据流的方式，而且能够像乐高一样模块化。

5.1.1 流的类型

流通常包含某种 I/O，而且它们能够通过其处理的类型分到相应的组。如下类型的流都出自 James Halliday 的 `stream-handbook` (<https://github.com/substack/stream-handbook/>)，他告诉你流适合做什么：

- 内置——许多 Node 的核心模块都实现了流接口；例如 `fs.createReadStream`。
- HTTP——除了网络技术的流，还有被设计处理其他网络技术的流。
- 解析器——以前很多解析器都使用流来实现。比较流行的三方模块包括 XML 和 JSON 解析器。
- 浏览器——事件驱动的 Node 流可以被扩展使用在浏览器，使用客户端代码提供功能。
- Audio——James Halliday 写了一些有流接口的声音模块。
- RPC（远程调用）——通过网络发送流是进程间通信的有效方式。
- 测试——有很多能够友好使用流的测试库和工具用来测试流本身。
- 控制、元以及状态是对流更加抽象的使用，而且模块纯粹地被设计用来操纵和管理流。

理解为什么流很重要的最好的方式就是想一下没有它们怎么处理数据。来让我们更详细地对比一下 Node 的同步、异步和流接口。

5.1.2 什么时候使用流

当使用 `fs.readFileSync` 同步读取一个文件的时候，程序将会被阻塞，所有的数据将会被读到内存中。使用 `fs.readFile` 将会阻止程序阻塞，因为它是异步方法，但是它仍然会将余留文件数据读到内存中。

倘若有一个方式可以告知 `fs.readFile` 去读取一个数据块到内存中，处理它，再去索取更多数据呢？这就要说到流了。

当处理大文件压缩、归档、媒体文件和巨大的日志文件等的时候，内存使用就成了问题。取代把剩余文件数据读到内存中，你可以使用 `fs.read` 配合一个合适的缓冲区，一次读取固定的长度。或许，你可以使用 `fs.createReadStream` 提供的流 API。图 5.1 演示了使用 `fs.createRead` 和使用 `fs.readFile` 读取剩余文件数据的对比。

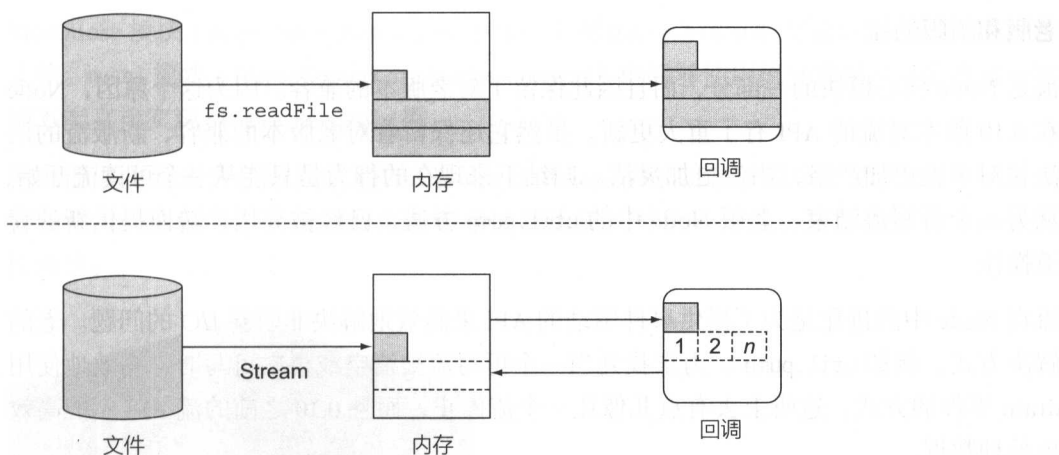


图 5.1 使用流式的 API 意味着 I/O 操作可能会使用更少内存

流被设计为异步的方式。相比将剩余文件数据一次性读进内存，还是值得读取一个缓冲的，期望的操作将会被执行，而且结果会被写到输出流。这种方式最接近 Node 惯用的方式。除此之外流使用老版本的 JavaScript 实现。使用 `fs.createReadStream`，提供了更多可扩展的方案，但是最终仅仅是用更好的对一些简单的文件系统操作封装了更好的 API。

Node 的流 API 是业界惯用的，当然流在计算机科学中发展很长时间了。这些历史会在下一节简单回顾，来告诉你流是从哪里来的，并且用在哪。

5.1.3 历史

那么流在哪里诞生？追溯历史，流在计算机科学中被用来处理跟它在 Node 中场景差不多的问题。例如，在 C 中表现文件的标准方式就是流。标准的 I/O 流在 Node 中也是可用的，并且能够被用来让命令程序在处理大文件数据时运行得更好。

传统地，流被用作实现高效率解析器。在 Node 中也是这样的：`node-formidable` (<https://github.com/felixge/node-formidable>) 模块被用来连接有效解析流数据，还有数据库模块，例如 `redis` (<https://npmjs.org/package/redis>)，使用流来实现与服务端的连接和对请求的解析。

如果对 UNIX 熟悉，或许已经注意到流了。如果使用过管道或者 I/O 重定向，那么你已经使用过流了。你可以从字面上认为 Node 的流就是 UNIX 中的管道数据，只不过是方法的而不是命令行的而已。下一节解释流是如何处理的，直到 0.10 版本有明显的改变。

老版和新版的流

流是 Node 核心模块的一部分，而且因此保留了对老版本的兼容。因为这个原因，Node 在 0.10 版本对流的 API 有了重大更新。虽然它还保留着对老版本的兼容，新版流的语法相对来说更加严格，当然更加灵活。归结下来现在的行为是只能从一个可读流开始，从另一个可写流结束。老版 Node 中的 `util.pump` 方法，已经被弃用，转而提供新的管道操作。

流在 Node 中的进化是为了提供事件驱动的 API 来高效地解决非阻塞 I/O 的问题。老的解决方式，例如 `util.pump`，力求找到当一个可写流被清空或再次可写时，高效地使用 `drain` 事件的方式。这听上去有点儿像让一个流停止，而且 0.10 之前的流 API 不能高效地处理数据。

现在，Node 的核心开发者已经看到人们讨论流的问题的类型，因此新的 API 能够更加丰富得益于原始的类。表 5.1 展示了 0.10 中可用的类。

表 5.1 新版流中可用类的总结

Name	User methods	Description
<code>stream.Readable</code>	<code>_read(size)</code>	用于在 I/O 上获取数据
<code>stream.Writable</code>	<code>_write(chunk, encoding, callback)</code>	用于在输出的目标写入数据
<code>stream.Duplex</code>	<code>_read(size)</code> , <code>_write(chunk, encoding, callback)</code>	一个可读和可写的流，例如网络连接
<code>stream.Transform</code>	<code>_flush(size)</code> , <code>_transform(chunk, encoding, callback)</code>	一个会以某种方式修改数据的双工流，没有输入数据要匹配输出数据的限制

学习了使用流将会获得更多的益处，尤其是当使用流实现三方模块的时候。下一节，我们将了解如何选择流行的第三方模块。

5.1.4 第三方模块中的流

流在 Node 中的主要用途是为 I/O 创建基于事件的 API、解析器、网络协议和数据库模块。用流实现一个网络协议会非常便利，试想一下可以通过管道调用 `gzip` 模块压缩数据会有多么容易。

同样，数据库操作大量数据会非常高效；相比将全部结果集中到一个数组，一次处理一个简单的条目可以使用流。

Mongoose 模块 (<http://mongoosejs.com/>) 有一个叫 QueryStream 的对象可以使用流来操作文档。mysql 模块 (<https://npmjs.org/package/mysql>) 也能够使用流处理结果, 虽然这个实现不能直接实现 stream.Readable 类。

你也能够找到更多使用流的地方。James Halliday 写的 baudio 模块 (见图 5.2) 用来生成音频流, 实现类似一些声音流能够通过管道混入其他声音流, 然后被标准的播放软件播放:

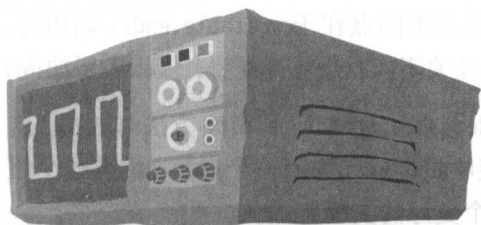


图 5.2 James Halliday (substack) 的 baudio 模块支持音频流的生成 (<https://github.com/substack/baudio>)

```
var baudio = require('baudio');

var n = 0;
var b = baudio(function (t) {
  var x = Math.sin(t * 262 + Math.sin(n));
  n += Math.sin(t);
  return x;
});
b.play();
```

当为你的 Node 工程选择一个网络或者数据库类库的时候, 我们强烈建议它支持流 API, 因为它会帮助你写更加优雅的代码, 同时也可能提高性能。

所有流类都继承自事件。这一特点的意义会在下一节来讨论。

5.1.5 流继承事件

每一个流模块的基类都会触发若干事件, 主要依赖基类是否可读, 可写或者都可以。事实是流继承自事件意味着你能够绑定标准事件管理流, 或者创建自定义事件来实现特定域的行为。

当使用 stream.Readable 实例, 可读事件是重要的, 因为它代表着流已经准备好调用 stream.read()。

为数据附上一个监听器会导致流行为像老的流 API，当数据可用时，数据通过数据监听器而不是通过调用 `stream.read()`。

错误事件详情在技巧 28 中。当流在接收数据遇到错误时将会触发。

结束事件表示流已经接收到了相当量的结束符，且不会再接收更多数据。还有一个关闭事件表示底层资源已经关闭，与结束事件不同，Node API 文档说明不是所有的流都会触发这个事件，因此首选应该是结束事件。

`stream.Writable` 类改变了表示流结束的语意。两者不同点在于 `writable.end()` 调用时，结束事件会触发，而关闭意味着底层 I/O 资源已经关闭，不会一直需要，依赖底层的流。

管道和非管道事件会在当通过一个流到 `stream.Readable.prototype.pipe` 方法时触发。这能够用于非管道情况下的适配流行为。监听器接收目标流作为第一个参数，因此这个值会在流变化的时候被检查。在技巧 37 会提及一个更好的实现方式。

关于本章的技术

本章的技术全部使用 `streams2` API。这是 Node 0.10 和 0.12 中新版 API 风格。如果你正在使用 Node 0.8，使用 `readable-stream` 模块（<https://github.com/isaacs/readable-stream>）向前兼容。

下一节你将会学习如何使用流处理现实场景下的问题。首先我们将会讨论一些 Node 内置的流，然后去创建新流并且测试它们。

5.2 内置流

Node 的核心模块是使用流模块实现的，因此可以直接开始使用流模块。后面的技术通过文件系统和网络流 API 介绍一些功能。

技巧 27 使用内置的流来实现静态 web 服务器

Node 核心模块通常提供流接口。相比同步的方式，使用它们可以更加高效地解决问题。

问题

你想要通过网络服务高效且支持大文件的发送一个文件到一个客户端。

解决方案

使用 `fs.createReadStream` 打开一个文件和流到客户端。通过管道处理结果数据 `stream.Readable`，就像处理压缩一样。

讨论

Node 的文件系统和网络操作的核心模块，`fs` 和 `net` 都提供了流接口。`fs` 模块拥有方法来自动创建流实例。使用流处理 I/O 问题会相当简单。

我们来理解为什么流很重要并且与不使用流的代码做对比，考虑如下的例子，使用 Node 核心模块实现简单的静态服务器：

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res) {
  fs.readFile(__dirname + '/index.html', function(err, data) { //
    if (err) {
      res.statusCode = 500;
      res.end(String(err));
    } else {
      res.end(data);
    }
  });
}).listen(8000);
```

即使这段代码使用非阻塞的 `fs.readFile` 方法，它需要用 `fs.createReadStream` 方法改进一下。原因是它会将剩余文件数据读进内存。小文件或许可以接受，但是当你不知道文件的大小呢？静态服务器通常可能提供大量二进制资源访问，因此需要一个适应性更强的解决方案。

下面是一个监听流的演示的例子。

例子 5.1 一个使用流的简单的静态 web 服务器

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res) {
  fs.createReadStream(__dirname + '/index.html').pipe(res);
}).listen(8000);
```

❶

❶ 数据通过管道的方式从一个文件输出到 Node 的 HTTP 请求响应。

例子使用了比第一版更少的代码，而且更加高效。现在替代一次性将剩余文件数据读入内存，值得提供一个缓冲区来发送到客户端。如果客户端连接较慢，网络流将会发送信号暂停 I/O 资源直到客户端准备好接收更多数据。这就像背压，而且也是使用流为程序带来的好处。

我们能够进一步改进例子。流不仅仅是高效、优雅的，它们也是可扩展的。静态服务通常使用 **gzip** 压缩文件。接下来的代码使用流的方式来给原先的例子添加 **gzip** 压缩。

例子 5.2 使用 **gzip** 压缩的静态 web 服务器

```
var http = require('http');
var fs = require('fs');
var zlib = require('zlib');

http.createServer(function(req, res) {
  res.writeHead(200, { 'content-encoding': 'gzip' });
  fs.createReadStream(__dirname + '/index.html')
    .pipe(zlib.createGzip())
    .pipe(res);
}).listen(8000);
```

❶

❷

❶ 设置头部以便让浏览器知道开启了 **gzip** 压缩。

❷ 两个管道的调用，分别用来压缩文件和把文件以流的方式输出到客户端。

现在如果你在浏览器打开 `http://localhost:8000` 并使用调试工具查看网络请求，你会看到内容已经转为使用 **gzip**。图 5.3 展示了运行例子的结果。

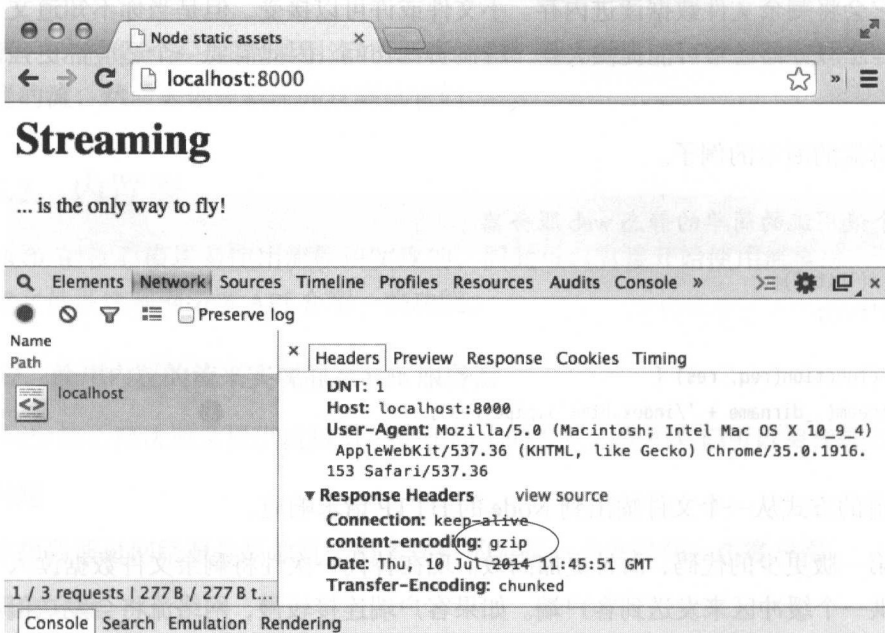


图 5.3 网络监测器确认内容是已经压缩的

这能够扩展为其他若干使用管道的方式。例如文件需要通过 HTML 模板引擎然后压缩。只需要记住一般的形式是可读的。

注意这个例子用来表明流如何工作，但并不是以实现一个生产环境的静态服务器。

现在你已经看到了充足的实例来说明流如何解决通用问题，是时候来看一下其他的部分：错误处理。

技巧 28 流的错误处理

流类集成子事件，这就意味着提供了标准明晰的错误处理。这个技术解释了如何处理流产生的错误。

问题

你想要获取一个流产生的错误。

解决方案

添加一个错误监听器。

讨论

事件的标准行为是当发生错误或者事件触发时会抛出一个异常，除非监听了错误事件。监听器的第一个参数是错误，是 Error 对象的后代。

下面通过事件监听来表明一个故意产生的错误。

例子 5.3 在流处理过程中捕获错误

```
var fs = require('fs');  
var stream = fs.createReadStream('not-found');  
  
stream.on('error', function(err) {  
  console.trace();  
  console.error('Stack:', err.stack);  
  console.error('The error raised was:', err);  
});
```

❶ 尝试打开的时候会引发一个错误。

❷ 使用事件的 API 来添加一个错误处理句柄。

这里我们企图打开一个不存在的文件❶，引起一个错误事件的触发。通过句柄传过来的错误对象❷通常还有额外信息帮助追查错误。举个例子，堆栈可能包含行信息，而且

`console.trace()` 能够生成完整的堆栈跟踪。监听时使用 `console.trace()` 会将 Node 核心模块 `events.js` 实现读流功能的地方跟踪到。这就意味着你能够准确知道在哪里触发了。现在你已经看过了一些 Node 的核心模块使用流，下一节探讨怎么通过第三方模块使用它们。

5.3 第三方模块和流

流在惯用的 Node 中很常见，所以在开源的 Node 项目中突然遍布流接口也不奇怪。下一个技巧，我们将会学习如何在流行的 Node 模块中使用流。

技巧 29 使用流的第三方模块

许多开源的开发者认为流很重要并且在他们的模块中使用。在如下内容中你将会学习到这些实现并且用它们更高效地解决问题。

问题

你想知道如何使用从 npm 下载第三方流模块。

解决方案

查看模块文档，或者从源代码弄清楚它如何实现流接口，如果是这样，那怎么来使用。

讨论

我们选择了三个流行的模块作为实现流接口的例子。这些流文档会给开发者使用流的好方案，并且会知道如何在你的项目中利用流。

下一节你将会发现在使用流行 web 框架 Express 中使用流的一些关键点。

在 Express 中使用流

Express web 框架 (<http://expressjs.com/>) 实际上是对 Node 的 http 核心模块提供了相对轻量的封装。它包括请求和响应对象。Express 用自己的方法和值优化这些对象，但是底层对象还是一样的。这意味着你学过的流的用法都可以在这复用。

来看一个简单的例子，Express 路由回调中使用 `res.send` 响应一些数据。

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
  res.send('hello world');
});
```

```
app.listen(3000);
```

res 对象实际上是一个 *response* 对象，它继承自 Node 的 `http.ServerResponse`。在技巧 27 中 HTTP 请求能够通过管道使用流。Express 集成了一种方式让 `res.send` 方法使用流，并且对于流你仍然能够使用管道。

例子 5.4 是一个 Express web 应用，运行了 Express 3 和使用管道的一个可写流内容。

例子 5.4 一个使用流的 Express 应用

```
var stream = require('stream');
var util = require('util');
var express = require('express');
var app = express();

util.inherits(StatStream, stream.Readable);

function StatStream(limit) {
  stream.Readable.call(this);
  this.limit = limit;
}

StatStream.prototype._read = function(size) {
  if (this.limit === 0) {
    // Done
    this.push();
  } else {
    this.push(util.inspect(process.memoryUsage()));
    this.push('\n');
    this.limit--;
  }
};

app.get('/', function(req, res) {
  var statStream = new StatStream(10);
  statStream.pipe(res);
});

app.listen(3000);
```

❶ 继承于 `stream.Readable` 来创建一个可读流，并且调用父类的构造函数。

❷ 用一些数据来响应请求，这里是发送一个表示 Node 进程内存用量的字符串。

❸ 使用标准的 `readable.pipe(writable)` 模式来把数据返回给浏览器。

自定义可读流、统计流、继承自 `stream.Readable` ①并且实现了 `_read` 方法，用来发送内存使用量②。当你想要创建一个可读流的时候必须要实现 `_read` 方法。当向浏览器发送响应数据时，流可以直接通过管道传给 `res` 对象③，不需要额外工作。

Express 3 中使用 `fs.createReadStream` 实现发送模块，像技巧 27 描述的。如下代码是发送的源代码里的：

```
SendStream.prototype.stream = function(path, options){
  // 待办：这都是不靠谱的，需要重构
  var self = this;
  var res = this.res;
  var req = this.req;

  // 使用管道
  var stream = fs.createReadStream(path, options);
  this.emit('stream', stream);
  stream.pipe(res);
}
```

它需要更多工作来修正内容，比如 HTTP 的响应头，但是这个代码片段表明使用内置的流 `fs.createReadStream` 能够支撑足够大的开源项目。

在 Mongoose 中使用流

为 MongoDB 数据库 (<http://www.mongodb.org/>) 开发的 Mongoose 模块 (<http://mongoosejs.com/>) 中有一个叫 `QueryStream` 的接口提供 Node 0.8 风格的流操作方式。在内部使用这个类允许用流的方式查询结果。如下的代码展示了用流来操作一个可写流：

```
User
  .where('role')
  .equals('admin')
  .stream()
  .pipe(writeStream);
```

这种模式——使用一个类来封装外部 I/O 流行为，然后通过简单的方法暴露流的调用，是 Node 核心模块采用的风格，在流行的三方模块中也这样。这在 `streams2` API 中更加简洁，可以通过简单的抽象类继承。

在 MySQL 中使用流

`mysql` 模块 (<https://npmjs.org/package/mysql>) 被 Node 开发者认为是相对低级的，例如 `Sequelize` (<http://www.sequelizejs.com/>) 对象关系映射 (ORM)。但是 `mysql` 模块本身不应该被低估，它支持流结果的暂停和恢复。这有一个演示基础 API 的例子：

```
var query = connection.query('SELECT * FROM posts');
query
.on('result', function(row) {
  connection.pause();
  processRow(row, function() {
    connection.resume();
  });
});
```

这个流 API 使用特定域事件名和领域事件。要暂停结果流，需要调用 `connection.pause`。这个信号通知底层 MySQL 连接，直到能接收更多数据之前，结果应该被暂停。

总结

在这个技巧中，我们看到了流行的第三方模块如何使用流，它们的共同特征就是处理 I/O，HTTP 和数据库连接是网络的或基于文件协议的，而且都能够处理网络连接和文件操作。通常，找到实现了流接口的 Node 网络框架或者数据库模块是个不错的主意，因为它们能够帮助你缩小问题，书写可读的优雅的代码。

现在已经学习了如何使用流，你或许已经想学习如何自己创建了。下一节有一个技术板块来使用每一个类，而且会展示如何继承它们。

5.4 使用流基类

Node 的技术流类提供模板来解决流最在行的这类问题。例如 `stream.Transform` 最适合解析数据，`stream.Readable` 是对低级 API 的完美封装。

下一个技巧解释了如何从流的基础类继承，并且从更进一步的技术板块来看如何使用每一个类的细节。

技巧 30 正确地从流的基类继承

Node 的流基类能够被用作新模块或者子类的起点。理解它们处理什么问题 and 如何正确地继承它们很重要。

问题

你想通过创建一个流 API 解决问题，但是不确定用哪个流的基类。

解决方案

决定哪个基类最适合解决手头上的问题，使用 `Object.prototype.call` 和 `util.inherits` 继承它。

讨论

Node 的流基类，已经在表 5.1 中总结，可以被用来实现自己的流类或模块。它们是抽象类，意味着在使用之前你必须实现它们。一般使用继承的方式实现。

所有的流基类都可以在流核心模块中找到。5 个基础类为 Readable、Writable、Duplex、Transform 和 PassThrough。从根本上说，流不是可读就是可写的，但是双工流都可以。这是很有意义的，你考虑一下 I/O 接口的行为，如一个网络链接是可读并且可写的。假设 ssh 只能发送数据，那么它就不是特别有用的了。

转换流是基于双工流的，但是也一定程度上改变了数据。一些 Node 内置的模块使用了转换流，因此从根本上来说很重要。其中一个例子就是 crypto 模块。

表 5.2 提供了一些线索来帮助你选择使用哪个基类。

表 5.2 选择一个流的基础类

Problem	Solution
你想要使用流 API 来包装一个底层的 I/O 数据源	Readable
你想要从一个程序中获取输出到其他地方使用，或者在程序中发送数据	Writable
你想要以某种方式解析数据并且修改它	Transform
你想要包装一个数据源，并且它也可以接收消息	Duplex
你想要从流中提取数据，从测试到分析都不修改它	PassThrough

从基类继承

如果你已经学会 JavaScript 继承，你可能习惯使用 `MyStream.prototype = new stream.Readable()` 继承流基类。这被认为是不好的做法，最好替换为用 ECMAScript 5 的 `Object.create` 模式。而且基类的构造器也必须运行，因为它提供了必要的启动代码。这个模式展示如下。

例子 5.5 从 stream.Readable 基类继承

```
var Readable = require('stream').Readable;
```

```
function MyStream(options) {  
  Readable.call(this, options);  
}
```

1

```
MyStream.prototype = Object.create(Readable.prototype, {  
  constructor: { value: MyStream }  
});
```

2

❶ 调用父类的构造函数，确保和它一样来传递任意的配置。

❷ 使用 `Object.create` 来正确设置原型链。

Node 包含一个工具类方法 `util.inherits`，可以用来替代 `Object.create`，但是两种方式都被开发者广泛使用。这个例子使用 `Object.create` 方法❶，你也可以看看 `util.inherits` 是如何实现的。

注意在例子 5.5 中，参数❷被传入 `Readable` 的构造器。很重要的一点是 Node 支持有一个标准的 `options` 配置流。如果是 `Readable`，`options` 如下：

- `highWaterMark`——停止读取底层数据源之前的内部缓冲数据的大小。
- `encoding`——触发缓冲数据自动编码。可能值包含 `utf8` 和 `ascii`。
- `objectMode`——允许流是一个流对象，而不是字节。

`objectMode` option 允许流处理 JavaScript 对象。技巧 31 提供了一个相关的例子。

总结

通过这个技巧你已经看了如何使用 Node 流基类创建自己的流实现。这里涉及使用 `util.inherits` 来创建类，而且使用 `.call` 调用原构造器。我们也覆盖这些类使用的 `options`。

正确地继承基类是一件事，但是实现一个流类呢？技巧 31 更详细地解释了可读基类，但是具体问题是它涉及实现了一个 `_read` 方法从底层数据源读取数据，而且又将它推送到一个内部队列管理起来。

技巧 31 实现一个可读流

可读流被用来为 I/O 源提供灵活的 API，也可以被用作解析器。

问题

你想要使用流 API 提供的高级接口封装 I/O 源。

解决方案

通过继承 `stream.Readable` 类实现一个可读流，同时创建一个 `_read(size)` 方法。

讨论

当围绕一个所需要的底层数据而实现一个自定义 `stream.Readable` 类是有用的。例如，我正在开发一个项目，客户端发送了 JSON 文件，包含数百万行的数据。我决定写一个简单的 `stream.Readable` 类来读取一个缓冲区，当一个新行出现，使用 `JSON.parse` 解析记录。

使用 `stream.Readable` 解析有换行符的记录如下展示。

例子 5.6 一个 JSON 行解析器

```
var stream = require('stream');
var util = require('util');
var fs = require('fs');

function JSONLineReader(source) {
  stream.Readable.call(this);
  this._source = source;
  this._foundLineEnd = false;
  this._buffer = '';

  source.on('readable', function() {
    this.read();
  }.bind(this));
}

util.inherits(JSONLineReader, stream.Readable);

JSONLineReader.prototype._read = function(size) {
  var chunk;
  var line;
  var lineIndex;
  var result;

  if (this._buffer.length === 0) {
    chunk = this._source.read();
    this._buffer += chunk;
  }

  lineIndex = this._buffer.indexOf('\n');

  if (lineIndex !== -1) {
    line = this._buffer.slice(0, lineIndex);
    if (line) {
      result = JSON.parse(line);
      this._buffer = this._buffer.slice(lineIndex + 1);
      this.emit('object', result);
      this.push(util.inspect(result));
    } else {
      this._buffer = this._buffer.slice(1);
    }
  }
};
```



```
var input = fs.createReadStream(__dirname + '/json-lines.txt', {
  encoding: 'utf8'
});
var jsonLineReader = new JSONLineReader(input);

jsonLineReader.on('object', function(obj) {
  console.log('pos:', obj.position, '- letter:', obj.letter);
});
```

9

- ❶ 通常要确保调用父类的构造函数。
- ❷ 当数据源准备好可以触发之后的 reads 事件时调用 read()。
- ❸ 从 stream.Readable 继承来创建一个可定制的新类。
- ❹ 所有的定制 stream.Readable 类都必须实现 _read() 方法。
- ❺ 当类准备好接收更多数据时，在源上调用 read()。
- ❻ 从 buffer 的开始截取第一行来获取一些文本进行解析。
- ❼ 无论何时当一个 JSON 记录解析出来时，触发一个“object”事件，对这个类来说是唯一的，但不是必需的。
- ❽ 把解析好的 JSON 发送回内部队列。
- ❾ 创建一个 JSONLineReader 的实例，传递一个文件流给它处理。

例子 5.6 使用了一个构造器函数，JSONLineReader ❶，继承自 stream.Readable ❸，来读取一个多行的 JSON 文件。JSONLineReader 的源数据也是一个可读流，因此也绑定了一个可读事件，因此 JSONLineReader 知道何时开始解析 ❷。

_read 方法 ❹检查缓冲区是否为空 ❺，如果是，从源读取更多数据添加到内部的缓冲区。然后当前行的索引递增，并且如果到达行尾，首行就从 buffer 中分割 ❻。一旦到达一个整行，解析器触发对象事件 ❼——类的用户能够绑定此事件接收每一行 JSON 数据。

当这个例子运行，一个文件的数据会通过这个类的实例。在内部，数据将被排队。当 source.read 被执行，最后的数据块将会被返回，因此 JSONLineReader 已经准备好的时候能够处理它。一旦足够的数据被读取或到达一个新行，然后结果会调用 this.push ❽收集起来。

一旦 this.push 被调用，stream.Readable 将会把排队的结果转发给一个消费流。这使得该流可以进一步通过管道进行可写流处理。在这个例子中 JSON 对象通过一个自定义事件触发。本例中的最后几行附加的事件侦听器监听处理结果 ❾。

`Readable.prototype._read` 的大小参数是 *advisory*。这意味着底层的实现可以用它来知道有多少数据获取，这让你可以不实现并不必要的。在之前的例子中，我们解析了整行，但一些数据格式可以以块的形式处理，在这种情况下，大小参数是很重要的。

在这个例子的原代码的基础上，我使用结果 JSON 对象来填充数据库。该数据也被重定向和 `gzip` 压缩到另一个文件。这种流容易编写，易于阅读。

例子 5.6 使用了字符串，但对于对象的例子呢？最多的流直接处理 I/O——文件、网络协议等，将使用原始字节或字符串流。但有时创建 JavaScript 对象流是有用的。例子 5.7 显示了如何安全地从 `stream.Readable` 继承，以及传递对象作为参数来设置一个流，并且作为 JavaScript 对象来进行处理。

例子 5.7 使用 `objectMode` 配置的流

```
var stream = require('stream');
var util = require('util');

util.inherits(MemoryStream, stream.Readable);

function MemoryStream(options) {
  options = options || {};
  options.objectMode = true;
  stream.Readable.call(this, options);
}

MemoryStream.prototype._read = function(size) {
  this.push(process.memoryUsage());
};

var memoryStream = new MemoryStream();
memoryStream.on('readable', function() {
  var output = memoryStream.read();
  console.log('Type: %s, value: %j', typeof output, output);
});
```

❶ 这个流应该都使用 `objectMode`，所以在这里设置，并且把剩余的设置参数传递给 `stream.Readable` 构造函数。

❷ 调用 Node 内置的 `process.memoryUsage()` 方法来创建一个对象。

❸ 给可读流添加一个监听器来跟踪什么时候流准备好可以输出数据，然后调用 `stream.read()` 来获取最新的数据。

在例子 5.7 中, `MemoryStream` 示例使用数据对象, 所以 `objectMode` 传递给可读构造作为一个选项^❶。然后 `process.memoryUsage` 被用来产生一些合适的的数据^❷。当这个类的实例发出可读^❸, 表示它已经准备好接收, 之后内存使用情况的数据将被记录到控制台读取。

当使用 `objectMode` 时, 流的底层行为发生改变, 用来除去内部缓冲区合并和长度检查, 并且读取和写入时忽略大小参数。

技巧 32 实现一个可写流

可写的流可用于输出数据到底层的 I/O。

问题

你想使用一个流接口 I/O 输出数据。

解决方案

继承 `stream.Writable` 并且实现一个 `_write` 方法向底层源数据发送数据。

讨论

正如你在技巧 29 看到的, 许多第三方模块提供网络服务和数据库流接口。跟随这一趋势是有利的, 因为它允许你的类使用管道的 API, 这有助于保持码块的复用和解耦。

你可能会只是希望简单实现一个可写流作为管道链, 或者实现不支持的 I/O 资源。在一般情况下, 所有你需要做的是正确地继承 `stream.Writable` ——更多的推荐使用的方式, 请参阅技巧 30, 然后添加一个 `_write` 方法。

所有的 `_write` 方法需要做的就是当数据被写入时调用提供的回调。下面的代码显示方法的参数和样本 `_write` 的实现:

```
MyWritable.prototype._write = function(chunk, encoding, callback) {❶  
  this.customWriteOperation(chunk, function(err) {❷  
    callback(err); //❸  
  });  
};
```

❶ `chunk` 参数是 `Buffer` 的一个实例或者是一个字符串。

❷ `customWriteOperation` 是你的类中自定义的写操作。它可以是异步的, 这样的话回调会在过后安全地调用。

❸ 如果发生了错误, 要调用 `Node` 内部代码提供的回调

一个 `_write` 方法提供了一个回调^❶, 你可以在写完成时调用。这允许 `_write` 是异步的。

在这里 `customWriteOperation` 方法^❷被简单地使用——在真正实现时，将执行基本 I/O。这可能涉及通过套接字访问数据库，或写入文件。提供给回调的第一个参数应该是一个错误^❸，允许在需要时 `_write` 传递错误。

节点的 `stream.Writable` 基类并不需要知道数据如何写，它只需要关心操作是成功还是失败。错误可以通过传递一个错误的对象回调报告调用者。这将导致错误的事件被触发。请记住这些数据流的基类，所以你通常应该添加一个监听器捕获错误并优雅地处理错误。

接下来的例子显示完整地实现了 `stream.Writable` 类。

例子 5.8 实现一个可写流的例子

```
var stream = require('stream');  
GreenStream.prototype = Object.create(stream.Writable.prototype, {  
  constructor: { value: GreenStream }  
});  
  
function GreenStream(options) {  
  stream.Writable.call(this, options);  
}  
  
GreenStream.prototype._write = function(chunk, encoding, callback) {  
  process.stdout.write('u001b[32m' + chunk + 'u001b[39m');  
  callback();  
};  
  
process.stdin.pipe(new GreenStream());
```

- ❶ 使用通常的继承模式来创建一个新的可写流类。
- ❷ 使用 ANSI 编码序列来给数据块添加绿色文本的标识。
- ❸ 当文本已经被发送到标准输出流时执行回调。
- ❹ 使用管道从输入到输出来把文本转换为绿色文本。

这个简短的示例将输入的文本颜色变为绿色。它可以通过 `node writable.js` 来运行，或将文本通过管道来使用，如 `cat file.txt | node writable.js`。

虽然这是一个简单的例子，但说明了实现流是很容易的，所以你应该考虑通过管道存储数据的工作来采用这种形式。

块与编码

编码参数只有当被用来代替缓冲区字符串时才会生效。字符串可以通过设置 `decodeStrings` 为 `false` 来使用，在实例化可写流时传递。

流并不总是处理缓冲区对象，因为一些实现了优化处理字符串，因此在某些情况下直接处理字符串可以更有效。

技巧 33 使用双工流转换和接收数据

双工流允许发送和接收数据。这个技巧告诉你如何创建自己的双工流。

问题

你想为需要可读写的 I/O 源创建一个流接口。

解决方案

继承 `stream.Duplex` 并实现 `_read` 和 `_write` 方法。

讨论

双工流是可写和可读的流，技巧 31 和技巧 32 已经解释了。因此，双工流需要继承 `stream.Duplex` 和实现 `_read` 和 `_write` 方法。参阅技巧 30 关于如何从流基类继承的解释。

例子 5.9 显示了读取和从标准输入、标准输出写入数据的 `stream.Duplex` 基类。它会提示输入数据，然后将其颜色 ANSI 转义码回写。

例子 5.9 一个双工流

```
var stream = require('stream');
```

```
HungryStream.prototype = Object.create(stream.Duplex.prototype, {  
  constructor: { value: HungryStream }  
});
```

```
function HungryStream(options) {  
  stream.Duplex.call(this, options);  
  this.waiting = false;  
}
```

```
HungryStream.prototype._write = function(chunk, encoding, callback) {  
  this.waiting = false;  
  this.push('u001b[32m' + chunk + 'u001b[39m');
```

```
    callback();  
  };  
  
  HungryStream.prototype._read = function(size) {  
    if (!this.waiting) {  
      this.push('Feed me data! > ');  
      this.waiting = true;  
    }  
  };  
  
  var hungryStream = new HungryStream();  
  process.stdin.pipe(hungryStream).pipe(process.stdout);
```

- ❶ 这个属性用来跟踪是否提示是否已经展示。
- ❷ `_write` 的实现把数据推送到内部的队列中，然后执行内部提供的回调函数。
- ❸ 当等待数据时，展示一个提示。
- ❹ 获取标准输入，用管道传给双工流，然后返回给标准输出。

在例子 5.9 中 `HungryStream` 类会显示一个提示，等待输入，然后返回 ANSI 颜色代码输入。要跟踪提示的状态，使用一个内部属性❶。该 `_write` 方法，将由 Node 自动调用，设置 `waiting` 属性为 `false`，说明输入已被接收，然后将带有颜色的数据推送到内部缓冲区。最后，就会自动传递给 `_write` 回调方法执行❷。

当这个类在等待数据时，`_read` 推送一条消息作为一条提示❸。通过管道使标准输入流经过 `HungryStream` 的实例，且回写到标准输出流来完成交互❹。

双工流的优势是处理管道之间的关系。一个更简单的方法是使用 `stream.PassThrough` 基类，因为在通过它时能够允许你将其插入管道中间和跟踪数据。图 5.4 从流的输入到输出，展示了数据流块如何通过双工流对象。

`stream.Duplex` 的一些外部实现提供了 `_write` 方法，却将 `_read` 方法留空。双工流纯粹的优势是通过管道的形式提高了其他流的表现。举个例子，由 Naomi Kyoto 提出的 `hiccups` 模块（<https://github.com/naomik/hiccups>）被用来模拟慢速和零散的底层 I/O 源的行为。当你正在写自动化测试的时候，使用流会更加便利。

双工流对管道数据的读到写甚至分析都是十分有用的。转换流是为了改变数据而特殊设计的；下一节介绍 `stream.Transform` 和 `_transform` 方法。

技巧 34 使用转换流解析数据

流被长期用来实现高效的解析器。`stream.Transform` 基类在 Node 中也能做同样的事。

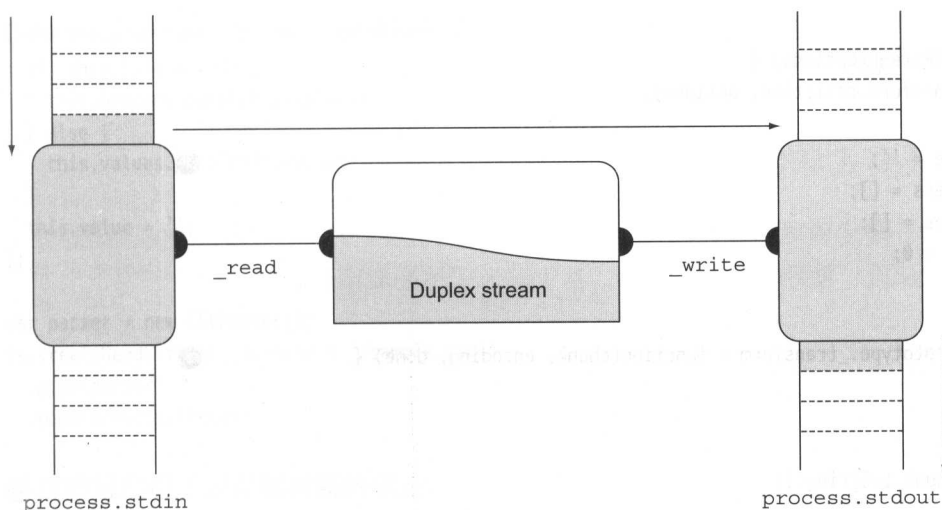


图 5.4 一个双工流

问题

你想使用流改变数据为另一种格式，且高效地管理内存。

解决方案

继承 `stream.Transform` 并实现 `_transform` 方法。

讨论

表面上看，转换流听起来有点像双工流。它们都能处理管道链中的关系。不同之处是转换流是转换数据，还是用 `_transform` 方法实现的。这个方法的特征很像 `_write` —— 它有三个参数，块、编码和回调。当数据被转换完成后会执行回调，允许转换流异步解析数据。

例子 5.10 展示了转换流解析很简单的 CSV 数据。CSV 格式数据除了使用逗号分割外，而不用额外的空格和引号，而且使用了 UNIX 的行结束符。

例子 5.10 使用转换流实现的一个 CSV 解析器

```
var fs = require('fs');
var stream = require('stream');
```

```
CSVParser.prototype = Object.create(stream.Transform.prototype, {
  constructor: { value: CSVParser }
});
```

```
function CSVParser(options) {  
  stream.Transform.call(this, options);
```

```
  this.value = '';  
  this.headers = [];  
  this.values = [];  
  this.line = 0;  
}
```

```
CSVParser.prototype._transform = function(chunk, encoding, done) {
```

```
  var c;  
  var i;
```

```
  chunk = chunk.toString();
```

```
  for (i = 0; i < chunk.length; i++) {  
    c = chunk.charAt(i);
```

```
    if (c === ',') {  
      this.addValue();
```

```
    } else if (c === '\n') {  
      this.addValue();
```

```
      if (this.line > 0) {  
        this.push(JSON.stringify(this.toObject()));  
      }
```

```
      this.values = [];  
      this.line++;
```

```
    } else {  
      this.value += c;  
    }  
  }
```

```
  done();
```

```
};
```

```
CSVParser.prototype.toObject = function() {
```

```
  var i;
```

```
  var obj = {};
```

```
  for (i = 0; i < this.headers.length; i++) {
```

```
    obj[this.headers[i]] = this.values[i];
```

```
  }
```

```
  return obj;
```

```
};
```



```
CSVParser.prototype.addValue = function() {  
  if (this.line === 0) {  
    this.headers.push(this.value);  
  } else {  
    this.values.push(this.value);  
  }  
  this.value = '';  
};  
  
var parser = new CSVParser();  
fs.createReadStream(__dirname + '/sample.csv')  
  .pipe(parser)  
  .pipe(process.stdout);
```

8

- ❶ 这些属性用于跟踪解析器的状态。
- ❷ `_transform` 的实现。
- ❸ 输入的数据转换为一个字符串，然后按照字符一个一个遍历。
- ❹ 如果字符是一个分号，添加之前收集的数据到内部的头部信息或者值信息的列表中。
- ❺ 如果字符是一个行结束符，记录先前收集的头部或者字段，然后使用 `push` 来发送一个数据字段的 JSON 版本给内部队列。
- ❻ 当处理完成后，执行 Node 提供的回调。
- ❼ 把头部和对应的最近行字段的内部数组转化为可以转化为 JSON 的对象。
- ❽ 头部信息应该是在第一行，否则最近收集的数据应该是一个数据字段的值。

解析 CSV 包括跟踪多个变量——当前值、文件头和当前的行号❶。要做这些，一个 `stream.Transform` 后代配合一些合适的属性能够使用。`_transform` 的实现❷是这个例子最复杂的部分。它接收到一个数据块，使用 `for` 循环一次遍历一个字母❸。如果字符是一个逗号，当前值就保存❹（如果有）。如果当前值是一个新行，当前行就被转为 JSON 表示❺。这个例子是同步的，因此在末尾执行 `_transform` 回调是安全的❻。`toObject` 方法包含头和值表示为内部 JavaScript 对象❼。

例子的最后一行创建了一个 CSV 数据的可读文件流，并且通过管道输送给 CSV 解析器，通过管道再次返回给标准输出，然后就可以看见结果了❽。这也可以使用管道输送给压缩模块，来直接支持压缩 CSV 文件，或者其他任何你可以想到的能用管道和流处理的事情。

例子不能实现真实的 CSV 文件包含的语意，但是可以演示使用流构建解析器却不是很简单，这仅取决于你的文件格式和协议。

现在你已经学会了如何使用基类。你或许想知道例子 5.10 中的参数是做什么的。下一节包含使用这些项来优化流的一些细节和一些进阶技术。

5.5 高级模式和优化

流基类接受各种选项定制它们的行为，而且其中一些选项可以用来调整性能。此部分具有技术来优化流，使用旧的流的 API，基于输入流适配和测试流。

技巧 35 流的优化

内置的流，用于构建自定义流的类允许配置的内部缓冲区的大小。很有必要知道如何优化这个值以获得所需的性能特性。

问题

你想从文件中读取数据，但对速度或内存性能有特定要求。

解决方案

优化流的缓冲区大小，以满足你的应用需求。

讨论

内置的流的函数获取缓冲器大小参数，允许性能的特性进行调整以适应特定的应用。`fs.createReadStream` 方法接受一个参数，可以包括一个 `bufferSize` 属性。该项被传递到 `stream.Readable`，所以它会控制用于可以暂时存储文件数据之前在其他地方使用的内部缓冲区。

由 `zlib.createGzip` 创建的流是 `streams.Transform` 的一个实例，并且 `Zlib` 的类创建了自己的内部缓冲器的对象来存储数据。控制该缓冲区的大小也是可能的，但是这一项的属性是 `chunkSize`。Node 的文档有一节关于优化 `zlib` 内存使用的内容，¹是在 `zlib/zconf.h` 头文件中，这是用来实现 `zlib` 本身低层次的部分源代码的内存使用情况的一节。

实际上将 Node 流推到不同的 CPU 相当困难——基于缓冲区大小的特性不同的 CPU。但要说明这个概念，我们提供了一个小的基准测试脚本，其中包括有关测量流性能的一些有趣的想法。接下来试图收集内存统计和经过时间。

¹参见“Memory Usage Tuning”—http://nodejs.org/docs/latest/api/all.html#all_process_memoryusage。

例子 5.11 流的基准测试

```
var fs = require('fs');
var zlib = require('zlib');

function benchStream(inSize, outSize) {
  var time = process.hrtime();
  var watermark = process.memoryUsage().rss;
  var input = fs.createReadStream('/usr/share/dict/words', {
    bufferSize: inSize
  });
  var gzip = zlib.createGzip({ chunkSize: outSize });
  var output = fs.createWriteStream('out.gz', { bufferSize: inSize });

  var memoryCheck = setInterval(function() {
    var rss = process.memoryUsage().rss;

    if (rss > watermark) {
      watermark = rss;
    }
  }, 50);

  input.on('end', function() {
    var memoryEnd = process.memoryUsage();
    clearInterval(memoryCheck);

    var diff = process.hrtime(time);
    console.log([
      inSize,
      outSize,
      (diff[0] * 1e9 + diff[1]) / 1000000,
      watermark / 1024].join(' '));
  });

  input.pipe(gzip).pipe(output);

  return input;
}

console.log('file size, gzip size, ms, RSS');

var fileSize = 128;
var zipSize = 5024;
```

```
function run(times) {
  benchStream(fileSize, zipSize).on('end', function() {
    times--;
    fileSize *= 2;
    zipSize *= 2;

    if (times > 0) {
      run(times);
    }
  });
}

run(10); 8((callout-streams-buffer-size-8))
```

- ❶ `hrtime` 用于获取当前的时间，精确到纳秒。
- ❷ 一个定时器的回调用来周期性地检查内存的使用，并且记录当前基准的最高使用值。
- ❸ 当输入结束时，收集统计数据。
- ❹ 打印内存使用情况和时间，把纳秒转换为毫秒。
- ❺ 把输入文件通过 `gzip` 实例压缩后输出。
- ❻ 每一个基准测试完成后会执行一下回调。
- ❼ 递归调用基准测试方法。
- ❽ 第一次调用基准测试方法，传入我想要它执行的次数。

这是一个很长的例子，但是它只是使用了一些 Node 内置的方法收集内存来设计不同的缓冲区大小。该基准测试流函数执行大部分的工作，并多次执行。它用 `hrtime` 记录了当前使用时间❶，会比 `Date.now()` 返回更加精准。输入流是 UNIX 目录文件，需要经过管道 `gzip` 流然后输出一个文件❺。然后基准流使用 `setInterval` 来运行内存使用❷。当输入流结束❸，存储器管道用法是基于之前和输入文件 `gzip` 压缩后的值计算的。

运行功能加倍输入文件的缓冲器和 `gzip` 缓冲器❻以显示对存储器的影响，并采取以读取流处理使用的时间。当输入文件的读数完成时，存储器使用量和经过时间将通过 `benchStream` 功能打印的输入文件❹，以便在运行时的基准完成轻易地调用。运行功能将被反复调用❼，取决于传递给的第一个参数❽。

需要注意的是 `process.hrtime` 已被用来准确地计算经过的时间。这种方法可以用于基准校对，因为它是精确的，并且还接受用于自动地计算所经过的时间的时间参数。

我运行这个程序有一个 20MB 的文件，试图在 `/usr/share/dict/words` 中生成有趣的结果，效果图已经显示在了图 5.5 中。

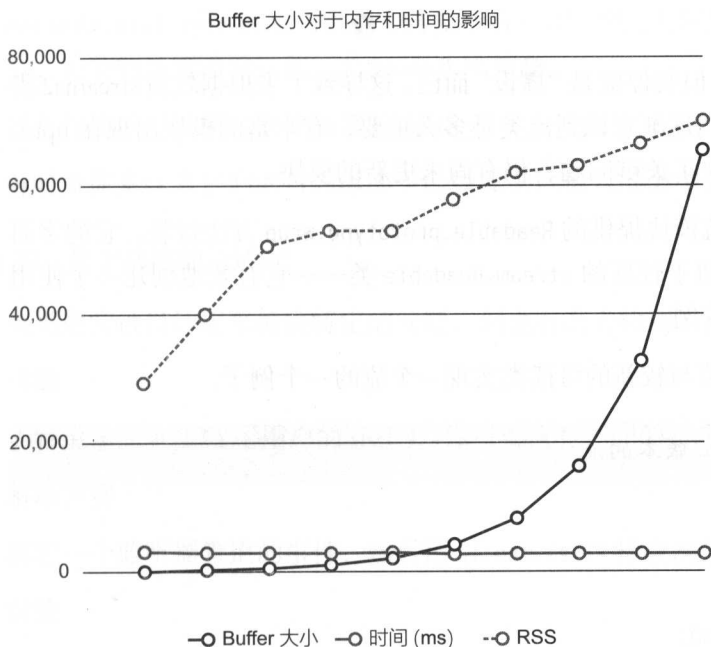


图 5.5 展示流使用内存的一幅图

我发现，当我用多个文件试验之后，结果表明，花费的时间影响远远小于内存的使用。这表明，它通常需要使用更小的缓冲器和更加保守的内存使用情况，但这个测试应该用重复的负载基准测试来看到 Node 处理这些缓冲区真正需要多长时间。

Node 有旧的 API，其中有用于流的、有用于暂停流不同的语义的。虽然新的 API 应该尽可能使用，也有可能一起使用旧的 API。下一个技巧演示如何使用旧 API 编写模块。

技巧 36 使用老的流 API

在 Node 0.10 (0.9.4)，流 API 发生了变化。使用该 API 编写代码可以通过封装它的行为作为 `stream.Readable` 类一起使用较新的 API。

问题

你想要用新 API 实现一个旧 API 风格的模块。

解决方案

使用 `Readable.prototype.wrap`。

讨论

旧的流 API 有读取和写入流，但暂停流是“虚设”而已。这导致了未根据较新 `streams2` 类不同的 API 的设计。随着人们逐渐意识到流类是多么重要，有丰富的模块出现在 npm。虽然新的 API 用旧的设计解决了关键问题，仍有尚未更新的模块。

幸运的是，旧的类可以使用流模块提供的 `Readable.prototype.wrap` 方法封装。它的字面封装的旧接口，使其行为类似于较新的 `stream.Readable` 类——它有效地创建一个使用旧的类作为其数据源的可读实例。

例子 5.12 显示了用旧 API 封装与较新的可读类实现一个流的一个例子。

例子 5.12 一个已经封装的旧版本的流

```
var stream = require('stream');
var Readable = stream.Readable;
var util = require('util');

util.inherits(MemoryStream, stream);

function MemoryStream(interval) {
  this.readable = true;

  setInterval(function() {
    var data = process.memoryUsage();
    data.date = new Date();
    this.emit('data', JSON.stringify(data) + '\n');
  }.bind(this), interval);
}

var memoryStream = new MemoryStream(250);
var wrappedStream = new Readable().wrap(memoryStream);

wrappedStream.pipe(process.stdout);
```

- ❶ 旧的 API 要求类从流模块中继承并且设置 `readable` 属性为 `true`。
- ❷ 用一些例子的值来触发 `data` 事件，确保使用字符串或者 `Buffer` 实例。
- ❸ 一个原始的流的实例必须被包装成为新的类的一个实例。
- ❹ 在这里，新的流使用管道发送数据给可写流的方式是和新的流 API 兼容的。

例子 5.12 给出了一个简单的继承自 Node 0.8 流模块的类。可读属性❶是旧的 API 的一部分，并表示一个可读流。另一个指标，这是一个传统的数据流是数据事件的❷。较新

`Readable.prototype.wrap` 方法^③来转换所有一切，使之与所述 `streams2` API 风格兼容。最后，包装的流管道输送到一个 Node 0.10 流^④。

现在你应该可以使用旧流与新的 API!

有时流需要改变它们的行为取决于已经提供该输入的类型。接下来的技巧正是这样做的。

技巧 37 基于功能的流适配

流类通常被设计用来解决特定的问题，但也有潜力检测正在使用的流定制它们的行为。

问题

当管道输送到 TTY (用户的 shell)，你想要流有不同的行为。

解决方案

绑定一个监听器管道的事件，然后用 `stream.isTTY` 检查流是否绑定到一个终端。

讨论

这种技术是适配流的行为，以它的环境中的一个特定的例子，但一般的方法可以适用于其他的问题。有时它是有用的，比如检测流是否正在写输出到一个 TTY 或别的东西，也许是一个文件，因为每一个不同的行为是可取的。例如，打印到一个 TTY 时，某些命令将使用 ANSI 色彩，但写入文件时，因为奇怪的字符会使结果乱码，这是不建议的。

Node 使得检测当前处理是否被连接至 TTY 只是使用 `process.stdout.isTTY` 和 `process.stdin.isTTY`。这些是从 Node 的源系统级绑定派生 (位于 `lib/tty.js`) 布尔值属性。

用于调整一个流的输出的策略是创建一个新 `stream.Writable` 类，并根据 `isTTY` 设置内部属性。然后侦听器添加到管道事件，基于新的管道流这样的传递的第一个参数的侦听器回调改变 `isTTY`。

例子 5.13 用两个类说明了这一点。第一，`MemoryStream` 继承自 `stream.Readable` 并生成基于 Node 的内存使用情况的数据。第二，`OutputStream`，监测它的必然性，因此它可以得出可读流的大约值，预计出什么样的输出。

例子 5.13 使用 `isTTY` 来适配流的行为

```
var stream = require('stream');
var util = require('util');
```

```
util.inherits(MemoryStream, stream.Readable);
util.inherits(OutputStream, stream.Writable);
```

```
function MemoryStream() {
  this.isTTY = process.stdout.isTTY;
  stream.Readable.call(this);
}

MemoryStream.prototype._read = function() {
  var text = JSON.stringify(process.memoryUsage()) + '\n';
  if (this.isTTY) {
    this.push('u001b[32m' + text + 'u001b[39m');
  } else {
    this.push(text);
  }
};

// A simple writable stream
function OutputStream() {
  stream.Writable.call(this);
  this.on('pipe', function(dest) {
    dest.isTTY = this.isTTY;
  }.bind(this));
}

OutputStream.prototype._write = function(chunk, encoding, cb) {
  util.print(chunk.toString());
  cb();
};

var memoryStream = new MemoryStream();

// Switch the desired output stream by commenting one of these lines:
//memoryStream.pipe(new OutputStream());
memoryStream.pipe(process.stdout);
```

❶ 设置一个内部的标识来记录希望哪种输出。

❷ 当打印到一个终端时，使用 ANSI 颜色。

❸ 当可写流和管道事件绑定时，修改目标的 isTTY 状态。

在内部，Node 使用 isTTY 适应 repl 模块和 readline 的界面的行为。例子 5.13 是跟踪 process.stdout.isTTY ❶ 的状态的例子，以确定原来的输出流是什么，然后复制后续目的地的值❸。当终端是一个 TTY，颜色都被用到了❷；否则使用纯文本输出代替。

流，跟任何其他东西一样，应该进行测试。接下来介绍的技巧将展示编写单元测试自己的流类的方法。

技巧 38 测试流

就像其他任何你写的，强烈建议你测试你的流。这个技巧说明了如何使用 Node 内置的断言模块测试的类继承自 `stream.Readable`。

问题

你写你自己的流类并且你想要写一个单元测试它。

解决方案

用一些合适的样本数据来驱动你的流类，然后调用 `read()` 或 `write()` 来收集结果，并比较预期的输出。

讨论

在 Node 的源本身和许多开源开发人员使用的进行流测试常见的模式是，驱动流中使用的样本数据进行测试，然后比较与预期值的结果。

想出适当的数据进行测试是最困难的部分。有时很容易创建一个文本文件，或 *fixture* 命名为测试，可以使用通过管道来驱动流。如果你正在测试一个面向网络的数据流，那么你应该考虑使用 Node 的网络或 `http` 模块创建产生合适的测试数据“mock”服务端。

例子 5.14 是 CSV 解析器的修改版本；它已经变成了一个模块，所以可以很容易地对其进行测试。例子 5.15 是关联的测试创建 `CSVParser` 的一个实例，然后通过它获取一些值。

例子 5.14 CSV 解析流

```
var stream = require('stream');
```

```
module.exports = CSVParser;
```

```
CSVParser.prototype = Object.create(stream.Transform.prototype, {  
  constructor: { value: CSVParser }  
});
```

```
function CSVParser(options) {  
  options = options || {};  
  options.objectMode = true;  
  stream.Transform.call(this, options);
```

```
  this.value = '';  
  this.headers = [];  
  this.values = [];  
  this.line = 0;
```

```
}
```

```
CSVParser.prototype._transform = function(chunk, encoding, done) {
```

```
  var c;
```

```
  var i;
```

```
  chunk = chunk.toString();
```

```
  for (i = 0; i < chunk.length; i++) {
```

```
    c = chunk.charAt(i);
```

```
    if (c === ',') {
```

```
      this.addValue();
```

```
    } else if (c === '\n') {
```

```
      this.addValue();
```

```
      if (this.line > 0) {
```

```
        this.push(this.toObject());
```

```
      }
```

```
      this.values = [];
```

```
      this.line++;
```

```
    } else {
```

```
      this.value += c;
```

```
    }
```

```
  }
```

```
  done();
```

```
};
```

```
CSVParser.prototype.toObject = function() {
```

```
  var i;
```

```
  var obj = {};
```

```
  for (i = 0; i < this.headers.length; i++) {
```

```
    obj[this.headers[i]] = this.values[i];
```

```
  }
```

```
  return obj;
```

```
};
```

```
CSVParser.prototype.addValue = function() {
```

```
  if (this.line === 0) {
```

```
    this.headers.push(this.value);
```

```
  } else {
```

```
    this.values.push(this.value);
```

```
  }
```

```
  this.value = '';
```

```
};
```

- ❶ 暴露类给外部，这样让其便于测试。
- ❷ 我们可以在一个类的实例上调用 `push()` 来测试这个方法。

该 `CSVParser` 类使用 `module.exports` 输出，因此它可以被单元测试加载❶。调用这个类的一个实例的 `_transform` 方法❷将在以后运行加载出口。接下来是一个简单的这个类的单元测试。

例子 5.15 测试 CSV 流解析器

```
var assert = require('assert');
var fs = require('fs');
var CSVParser = require('./csvparser');

var parser = new CSVParser();
var actual = [];

fs.createReadStream(__dirname + '/sample.csv')
  .pipe(parser);

process.on('exit', function() {
  actual.push(parser.read());
  actual.push(parser.read());
  actual.push(parser.read());

  var expected = [
    { name: 'Alex', location: 'UK', role: 'admin' },
    { name: 'Sam', location: 'France', role: 'user' },
    { name: 'John', location: 'Canada', role: 'user' }
  ];

  assert.deepEqual(expected, actual);
});
```

❶

❷

❸

❹

- ❶ 添加一个 `exit` 事件监听器，当流完成后，执行测试。
- ❷ 调用流的 `read()` 方法来收集数据。
- ❸ 创建一个数组来存放期望的值。
- ❹ 使用 `assert.deepEqual` 来正确地对比运行的值和期望的值。

一个 fixture 文件，`sample.csv`，已用于管道数据到 `CSVParser` 实例。然后 `assert.deepEqual` 方法很容易地比较实际与预期。

侦听器绑定到出口❶，因为我们要等待流运行断言之前完成处理数据。然后数据被从分析器读取❷和推送到一个数组来检查断言❸的预期值❹。本模式用于 Node 自身的数据流测试，并且是像 Mocha 和 Node 测试框架一样的轻量。

5.6 总结

在本章中，你已经看到了内置的流 API 是如何工作的，如何利用 Node 提供的基类来创建新的流，以及如何使用一些更先进的技术来构建类与流。正如你在技巧 36 所看到的，建立新的数据流从底层正确继承类，不要忘记来测试这些流！想了解更多有关测试内容，请参考技巧 38。

正如你所看到的，流的一些新用途，如 substack 的 `baudio` 模块 (<https://github.com/substack/baudio>) 充分说明声音流。也有两个流的 API：原始 Node 0.8 及较低版本的 API，和较新的 `streams2` 的 API。向前兼容性是通过读取流模块 (<https://github.com/isaacs/readable-stream>) 支持，而通过封装流实现向后兼容 (36)。

流工作的很大一部分是处理文件。在下一章中，我们将看看文件系统处理的细节。

文件系统：通过异步和同步的方法 处理文件

本章概要

- 理解 fs 模块及其组件
- 使用配置文件及文件描述
- 使用文件锁技术
- 递归文件操作
- 编写文件数据库
- 监听文件及文件夹

就像前一个章节我们看到的，Node 的核心模块通常是低级别的 API。这使得各种创意及高级别模块，例如 Web 框架、文件解析器，以及命令行工具，以第三方模块的形式存在。fs 或者说文件系统模块也是这样。

fs 模块通过以下方式允许开发者与文件系统交互：

- POSIX 文件 I/O
- 文件流

- 批量文件 I/O
- 文件监控

fs 模块相比较其他 I/O 模块（如 net 以及 http）比较特殊，它不但有异步接口，还有同步的接口。这意味着它提供了一种机制，从而实现同步 I/O。文件系统有同步的接口很大部分是因为 Node 自己的内部工作，也就是模块系统以及 require 方法的同步行为。

本章的目的是要告诉你一些技巧，和不同的复杂程度的例子，来与文件系统模块工作时使用。我们将会学到以下内容。

- 同步及异步的方法加载配置文件
- 使用文件描述
- 使用文件锁技术
- 递归文件操作
- 编写文件数据库
- 监听文件及文件夹的变化

但在此之前，让我们先看下你能够过文件系统接口来做什么。

6.1 fs 模块概述

fs 模块包含常规的 POSIX 文件操作的封装，以及批量操作、流和监听操作。它还有许多操作的同步接口。让我们看一下不同的组件。

6.1.1 POSIX 文件系统包装器

文件系统接口主要的方法是对标准 POSIX 文件 I/O 调用的封装。这些方法有着相同的名字，比如，readdir 在 Node 中有个对应的 fs.readdir 方法。

```
var fs = require('fs');
fs.readdir('/path/to/dir', function (err, files) {
  console.log(files); // [ 'fileA', 'fileB', 'fileC', 'dirA', 'etc' ]
});
```

表 6.1 展示了在 Node 中支持的 POSIX 文件方法及其功能描述。

表 6.1 node 支持的 POSIX 文件方法

POSIX 方法	fs 方法	描 述
rename(2)	fs.rename	改变文件名称

续表

POSIX 方法	fs 方法	描 述
truncate(2)	fs.truncate	截断或者扩展文件到指定的长度
ftruncate(2)	fs.ftruncate	和 truncate 一样，但将文件描述符作为参数
chown(2)	fs.chown	改变文件的所有者以及组
fchown(2)	fs.fchown	和 chown 一样，但将文件描述符作为参数
lchown(2)	fs.lchown	和 chown 一样，但不解析符号链接
chmod(2)	fs.chmod	修改文件权限
fchmod(2)	fs.fchmod	和 chmod 一样，但将文件描述符作为参数
lchmod(2)	fs.lchmod	和 chmod 一样，但不解析符号链接
stat(2)	fs.stat	获取文件状态
lstat(2)	fs.lstat	和 stat 一样，但返回信息是关于符号链接而不是它指向的内容
fstat(2)	fs.fstat	和 stat 一样，但将文件描述符作为参数
link(2)	fs.link	创建一个硬链接
symlink(2)	fs.symlink	创建一个软链接
readlink(2)	fs.readlink	读取一个软链接的值
realpath(2)	fs.realpath	返回规范的绝对路径名
unlink(2)	fs.unlink	删除文件
rmdir(2)	fs.rmdir	删除文件目录
mkdir(2)	fs.mkdir	创建文件目录
readdir(2)	fs.readdir	读取一个文件目录的内容
close(2)	fs.close	关闭一个文件描述符
open(2)	fs.open	打开或者创建一个文件用来读取或者写入
utimes(2)	fs.utimes	设置文件的读取和修改时间
futimes(2)	fs.futimes	和 utimes 一样，但将文件描述符作为参数
fsync(2)	fs.fsync	同步磁盘中的文件数据
write(2)	fs.write	写入数据到一个文件
read(2)	fs.read	读取一个文件的数据

POSIX 方法提供了很多文件操作的低级别的接口。比如这里我们用了几个同步的 POSIX 方法来把数据写到文件，然后再从文件中把数据读出来。

```
var fs = require('fs');
var assert = require('assert');
```

```
var fd = fs.openSync('./file.txt', 'w+');  
var writeBuf = new Buffer('some data to write');  
fs.writeFileSync(fd, writeBuf, 0, writeBuf.length, 0);  
  
var readBuf = new Buffer(writeBuf.length);  
fs.readSync(fd, readBuf, 0, writeBuf.length, 0);  
assert.equal(writeBuf.toString(), readBuf.toString());  
  
fs.closeSync(fd);
```

❶

❷

❸

❹

❺

❻

❼

- ❶ 打开或者创建 file.txt 用于写或者读（w+）。
- ❷ 创建一个数据 buffer 用于写入。
- ❸ 把 buffer 写入到文件中。
- ❹ 创建一个空的 buffer，大小和写入的 buffer 一样。
- ❺ 使用存储在文件中的数据填充 buffer。
- ❻ 断言写入的 buffer 和读取的 buffer 的数据是一致的。
- ❼ 关闭文件。

当要读写文件的时候，通常不需要使用这么低级别的接口，可以使用流或者大容量文件 I/O。

6.1.2 流

fs 模块通过 fs.createReadStream 以及 fs.createWriteStream 提供了流接口。fs.createReadStream 返回一个可读流，而 fs.createWriteStream 返回的是可写流。流接口可以通过 pipe 连接其他的流。比如，这里有一个用流来拷贝文件的简单例子：

```
var fs = require('fs');  
var readable = fs.createReadStream('./original.txt');  
var writeable = fs.createWriteStream('./copy.txt');  
readable.pipe(writeable);
```

❶

❷

❸

- ❶ 打开 original.txt 开始准备读取。
- ❷ 使用新数据创建或者覆盖 copy.txt。
- ❸ 当从 original.txt 读取数据时，将数据写入到 copy.txt。

文件流在你想一次性处理数据或者链接数据源的时候很有用。关于流的详细介绍，请查看第 5 章。

6.1.3 批量文件操作

文件系统接口还包括一些批量的方法来读写或者追加文件。

批量方法在你想把文件加载到内存或者一次性写入文件时很有用。

```
var fs = require('fs');
fs.readFile('/path/to/file', function (err, buf) {
  console.log(buf.toString());
});
```

❶

❶ 整个文件内容转成 buffer，放在 buf 变量中。

6.1.4 文件监视

fs 模块还提供了一些机制来监控文件（fs.watch 和 fs.watchFile）。这在你想知道一个文件是否改变的时候很有用。fs.watch 通过底层操作系统来通知文件改变，这非常高效。但 fs.watch 会比较难用，或者不能用在网络磁盘行。在这些情况下，可以使用相对低效的 fs.watchFile 方法。

我们将在本章后面看到更多文件监控相关的技术。

6.1.5 同步的替代方案

Node 的同步方法非常明显，在每个同步方法后有个首字母大写的 Sync，为了和异步的方法区分。同步的方法可用于所有的 POSIX 及批量接口。一些例子包括 readFileSync、statSync，以及 readdirSync。Sync 告诉你这个方法会阻塞你的单线程 Node 进程直到它结束。一般而言，同步方法应该在第一次初始化你的应用时使用，而不能在回调中使用。

```
var fs = require('fs');
var http = require('http');
fs.readFileSync('./output.dat');
```

❶

```
http.createServer(function (req, res) {
  fs.readFileSync('./output.dat');
}).listen(3000);
```

❷

❶ 同步方法的良好实践，在应用的初始化时使用。

❷ 同步方法的错误使用，在每一次请求中会阻塞服务器直到文件读取完。

当然这个规则也有例外，但理解同步的方法所带来的性能上的问题非常重要。

测试服务性能

我们如何知道在 Web 服务器处理请求的时候同步执行会慢？一个好的方法是通过 ApacheBench (<http://en.wikipedia.org/wiki/ApacheBench>) 来测试。我们前面的例子显示性能 2 倍下降，在每次同步请求 10MB 的文件时，而不是在应用程序设置缓存。下面是在该测试中所使用的命令

```
ab -n 1000 -c 100 "http://localhost:3000"
```

在快速浏览之后，我们现在已经准备好进入一些你会在与文件系统打交道时使用的技巧。

技巧 39 读取配置文件

把配置文件放在单独的文件中很有用，特别是那些应用运行在多个环境中（例如开发环境、预发环境以及生产环境）。在这个技巧，你会学到如何加载配置文件的来龙去脉。

问题

你的应用将配置文件保存在单独的文件中，并且在它启动的时候依赖这个文件。

解决方案

通过同步的文件系统方法在系统初始化的时候加载配置文件。

讨论

同步 API 通常的使用场景是在应用启动的时候加载配置文件或者其他数据。假设我们有一个简单的 JSON 格式的配置文件：

```
{
  "site title": "My Site",
  "site base url": "http://mysite.com",
  "google maps key": "92asdfase8230232138asdfasd",
  "site aliases": [ "http://www.mysite.com", "http://mysite.net" ]
}
```

先来看看我们如何异步地来做这件事，这样你可以看出不同。比如，doThisThing 这个方法依赖我们的配置文件上的数据，通过异步的方法是这个样子的：

```
var fs = require('fs');
fs.readFile('./config.json', function (err, buf) {
  if (err) throw er;
  var config = JSON.parse(buf.toString());
  doThisThing(config);
```

①

②

```
})
```

❶ 当没有这个配置文件时应用无法运行，我们仅会抛出异常，Node 进程会退出和打印跟踪堆栈。

❷ 我们拿到一个 buffer，将其转为字符串，然后解析成为 JSON。

这样是可行的而且可能是期望的，但是同样造成了所有依赖配置文件的方法都必须嵌套在一层回调中。这使得代码变得丑陋。通过使用同步的版本，我们可以更简洁地实现相同的结果。

```
var fs = require('fs');
var config = JSON.parse(fs.readFileSync('./config.json').toString());
doThisThing(config);
```

❶

❶ 同步的方法如果有错误，会自动抛出。

使用同步方法的一个特点是，一旦有错误发生，它将会抛出异常。

```
var fs = require('fs');
try {
  fs.readFileSync('./some-file');
}
catch (err) {
  console.error(err);
}
```

❶

❷

❶ 同步的错误可以使用标准的 try/catch 块来捕获。

❷ 处理错误。

使用 require 的注意事项

在 Node 中我们能使用 require JSON 文件把它作为一个模块，那样你的代码会更简短：

```
var config = require('./config.json');
doThisThing(config);
```

但这种方法有个警告。模块会被全局缓冲，所以假如我们有另外一个文件也加载了 config.json 并且修改了它，那么这个会影响到整个系统中其他加载了这个文件的模块。因此，当你想要修改对象的时候建议使用 readFileSync。假如你选择使用 require，那么把对象看作是只读的，否则你会面临很难追踪的 bug。你可以显式地通过 Object.freeze 来冻结一个对象。

这和异步方法不同，异步方法把 `error` 对象作为回调的第一个参数：

```
fs.readFile('./some-file', function (err, data) {  
  if (err) {  
    console.error(err);  
  }  
});
```

❶

❷

❶ 异步的错误可以在回调方法中作为第一个参数来进行处理。

❷ 处理错误。

在例子中，我们更希望当没能加载到文件的时候造成系统崩溃，但有些时候你希望处理同步的错误。

技巧 40 使用文件描述

假如你没有使用过文件描述，那么在一开始会比较困惑。这个技巧介绍了如何在 Node 中使用它们。

问题

希望通过文件描述读写文件。

解决方案

使用 Node 的 `fs` 文件描述模块。

讨论

文件描述是在操作系统中管理的在进程中打开文件所关联的一些数字或者索引。操作系统通过指派一个唯一的整数给每个打开的文件用来查看关于这个文件的更多信息。

虽然这个模块名字中有个 *file*，但它能做的远超过常规的文件。文件描述可以指向目录、管道、网络套接字以及常规文件。Node 可以理解这一些底层比特数的含义。大多数进程有一组标准的文件描述，如表 6.2 所示。

表 6.2 常规文件描述

Stream	File descriptor	Description
stdin	0	标准输入
stdout	1	标准输出
stderr	2	标准错误

在 Node 中，我们通常希望 stdout 打印日志的时候习惯用 `console.log` 语法糖：

```
console.log('Logging to stdout')
```

假如使用全局的 `process` 对象，我们可以更明确地达到同样的目的。

```
process.stdout.write('Logging to stdout')
```

但这里还有一种方法，很少会用到。`fs` 模块中有一些方法将文件描述作为第一个参数。我们能通过 `fs.writeFileSync` 写入文件描述 1 或者 `stdout`。

```
fs.writeFileSync(1, 'Logging to stdout')
```

同步日志

`console.log` 与 `process.stdout.write` 实际上是同步的方法，所提供的 TTY 是一个文件流。

一个文件描述是 `open` 以及 `openSync` 方法调用返回的一个数字。

```
var fd = fs.openSync('myfile', 'a');  
console.log(typeof fd == 'number');
```

❶

❶ 返回 `true`。

在文件系统的文档中，有很多的方法来处理文件描述。

更加有趣的文件描述符使用方法是当你继承父进程或者派生子进程的时候。我们会在后面章节讨论 `child process` 的时候再来看。

技巧 41 使用文件锁

文件锁在你需要协同多个进程同时访问一个文件并且要保证文件的完整性以及数据不能丢失的时候很有用。在这个技巧中，我们会看到如何实现一个文件锁模块。

问题

你想锁住文件来防止多个进程篡改该文件。

解决方案

通过 Node 的内置模块创建一个文件锁机制。

讨论

在单线程的 Node 进程中，文件锁通常不需要你操心。但在有些情况下会有其他的进程也要访问相同的文件，或者一个 Node 进程的集群要访问相同的文件。

在这些情况下，会有竞争并且数据丢失发生（更多信息参考 <http://mng.bz/yTLV>）。大多数操作系统提供了强制锁（在内核级别执行）以及咨询锁（非强制，只在涉及到的进程订阅了相同的锁机制）。咨询锁如果可能的话，通常优先选择，因为强制锁太重并且比较难解锁（<https://kernel.org/doc/Documentation/filesystems/mandatory-locking.txt>）。

使用第三方模块实现文件锁

Node 本身不能实现直接锁住一个文件（无论是强制锁还是咨询锁）。但咨询锁可以通过调用 syscalls，比如 flock（<http://linux.die.net/man/2/flock>）实现，它在第三方模块是可用的（<http://github.com/baudehlo/node-fs-ext>）。

除了通过 flock 直接锁住一个文件，你还可以使用锁文件。锁文件是普通的文件或者文件夹，它的存在表明其他资源正在被使用并且不能被篡改。锁文件的创建必须是原子性的从而避免冲突。作为咨询锁，所有的进程在锁文件存在时必须遵守相同的规则。图 6.1 展示了这一点。

举个例子，我们有个叫 config.json 的文件可能被很多进程同时修改。为了避免数据丢失或者损坏，那个正在修改文件的进程可以创建一个 config.lock 文件并在修改完成之后删除这个文件。其他所有的进程必须在做任何修改前检查锁文件是否存在。

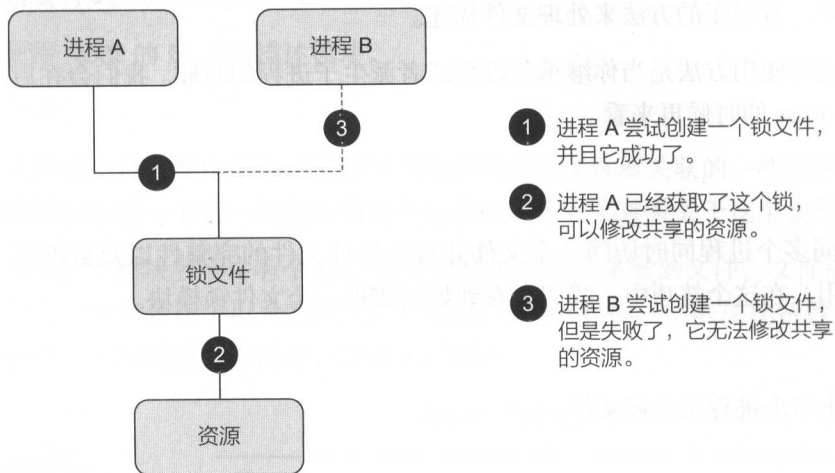


图 6.1 进程协作间使用一个锁文件来判断资源锁定

Node 提供了一些方法来实现这个技术。我们来看几个方法。

- 使用独占标记创建锁文件
- 使用 `mkdir` 创建锁文件

我们先来看通过独占标记的方法创建锁文件。

使用独占标记创建锁文件

`fs` 模块为所有需要打开文件的方法（比如 `fs.writeFile`、`fs.createWriteStream`，以及 `fs.open`）提供了一个 `x` 标记。这告诉操作系统这个文件应该以独占模式打开（`O_EXCL`）。当使用这个方法时，若这个文件存在，文件不能被打开。

```
fs.open('config.lock', 'wx', function (err) {  
  if (err) return console.error(err);  
  
});
```

❶
❷
❸

- ❶ 以可执行、可写入的模式打开。
- ❷ 任何失败，包括文件已存在。
- ❸ 安全地修改 `config.json`。

打开文件时用组合标记

当你打开文件时，可以用很多组合标记；可以查看 `fs.open` 的文档来看所有的组合列表：http://nodejs.org/api/fs.html#fs_fs_open_path_flags_mode_callback。

希望当另外一个进程创建了一个锁文件时，我们不能打开这个文件。因为我们不想在其他进程正在修改这个文件时去篡改这个文件。所以说独占锁机制在这个情况下很有用。但最好把当前进程的进程号写到这个锁文件中而不是写一个空文件，这样，当有异常发生时，我们知道最后拥有这个锁的进程。

```
fs.writeFile('config.lock', process.pid, { flags: 'wx' },  
  function (err) {  
    if (err) return console.error(err);  
  
  });
```

❶
❷
❸

- ❶ 如果不存在锁文件的话，写入 PID 来锁住文件。
- ❷ 任何失败，包括文件已存在。
- ❸ 安全地修改 `config.json`。

通过 mkdir 创建锁文件

当锁文件在网络磁盘上时独占模式可能不能正常工作，因为一些系统在网络磁盘上并不识别 `O_EXCL` 标记。要绕开这个问题，另外一个策略是把锁文件创建成一个文件夹。`mkdir` 是一个原子性的操作（没有并发），很好地支持跨平台，并且在网络磁盘上也能很好地运行。当目录已经存在时 `mkdir` 方法会失败。这个情况下，PID 可以写入这个目录中的一个文件。

```
fs.mkdir('config.lock', function (err) {  
  if (err) return console.error(err);  
  fs.writeFile('config.lock/'+process.pid, function (err) {  
    if (err) return console.error(err);  
  
  });  
});
```

- ❶ 无法创建目录。
- ❷ 标识 PID 已经锁住以便进行调试。
- ❸ 安全地修改 `config.json`。

创建一个锁文件模块

到现在为止，我们已经讨论了几个创建锁文件的方法，我们还需要一个方法在结束操作时删除它们。而且，作为一个好的锁文件使用者，我们应该在进程退出时删除所有的锁文件。这些方法可以封装在一个简单的模块中。

```
var fs = require('fs');  
var hasLock = false;  
var lockDir = 'config.lock';  
  
exports.lock = function (cb) {  
  if (hasLock) return cb();  
  
  fs.mkdir(lockDir, function (err) {  
    if (err) return cb(err);  
    fs.writeFile(lockDir+'/'+process.pid, function (err) {  
      if (err) console.error(err);  
      hasLock = true;  
      return cb();  
    });  
  });  
}
```



```
exports.unlock = function (cb) {  
  if (!hasLock) return cb();  
  
  fs.unlink(lockDir+'/'+process.pid, function (err) {  
    if (err) return cb(err);  
    fs.rmdir(lockDir, function (err) {  
      if (err) return cb(err);  
      hasLock = false;  
      cb();  
    });  
  });  
}  
  
process.on('exit', function () {  
  if (hasLock) {  
    fs.unlinkSync(lockDir+'/'+process.pid);  
    fs.rmdirSync(lockDir);  
    console.log('removed lock');  
  }  
});
```

- ❶ 定义一个方法来获取一个锁。
- ❷ 已经获取了一个锁。
- ❸ 无法创建一个锁。
- ❹ 把 PID 写入到目录中以便调试。
- ❺ 如果无法写 PID，并非世界末日：打印日志，然后继续运行。
- ❻ 锁创建了。
- ❼ 定义一个方法来释放锁。
- ❽ 没有需要解开的锁。
- ❾ 如果还有一个锁，在退出之前同步删除掉。

这是如何使用它的例子：

```
var locker = require('./locker');  
  
locker.lock(function (err) {  
  if (err) throw err;  
  
  locker.unlock(function () { });  
})
```

- ❶ 尝试获取一个锁。
- ❷ 在这里进行修改。
- ❸ 当完成后释放锁。

要了解更多实现独占模式的方法，可以去看第三方模块 `lockfile` (<https://github.com/isaacs/lockfile>)。

技巧 42 递归文件操作

是不是有时候需要像 `rm -rf` 那样删除一个目录以及它下面的所有子目录？或者创建一个目录以及它所有的中间目录？还是从多层目录下找到一个指定文件？递归文件操作非常有用也很容易出错，特别是异步操作时。但掌握如何操作它们对掌握事件驱动编程是个好的练习。在这个技巧中，我们将会通过创建一个查询多层目录的模块来深入理解递归文件操作。

问题

你希望从一个多层的目录中查找一个文件。

解决方案

使用递归以及文件系统模块。

讨论

当一个任务涉及到多层目录，事情就变得有趣了，尤其在一个异步的世界中。你可以用简单的 `fs.mkdir` 来模拟命令行的 `mkdir` 方法，但像 `mkdir -p`（可以创建中间目录），你必须考虑递归。这意味着要解决我们的问题就要先解决更小的相同的问题（“Recursion (computer science)”：[http://en.wikipedia.org/wiki/Recursion_\(computer_science\)](http://en.wikipedia.org/wiki/Recursion_(computer_science))）。

在这个例子中，我们会写一个查找模块。我们的查找模块可以依次从指定的开始路径查找文件（就像 `find /start/path -name='file-in-question'`），并把查找到的结果放入数组中。

我们有这样一个目录树：

```
root
├── dir-a
│   ├── dir-b
│   │   ├── dir-c
│   │   │   └── file-e.png
│   │   ├── file-c.js
│   │   └── file-d.txt
```

```
|   |—— file-a.js  
|   |—— file-b.txt
```

从根对模式 `/file.*/` 的查找会返回给我们下面的结果：

```
[ 'dir-a/dir-b/dir-c/file-e.png',  
  'dir-a/dir-b/file-c.js',  
  'dir-a/dir-b/file-d.txt',  
  'dir-a/file-a.js',  
  'dir-a/file-b.txt' ]
```

那么我们从何开始？fs 模块提供了一些我们需要的基本的方法。

- `fs.readdir/fs.readdirSync`——通过输入的地址参数列出包括目录的所有文件。
- `fs.stat/fs.statSync`——返回指定路径的文件信息，无论该路径是文件还是目录。

我们的模块会暴露同步的（`findSync`）以及异步的（`find`）两个实现。`findSync` 会像其他的同步方法一样阻塞进程，比对应的异步方法更快，但在碰到很大的文件目录时执行失败（因为 JavaScript 还没有合适的尾调用：<https://people.mozilla.org/~jorendorff/es6-draft.html#sec-tail-position-calls>）。

为什么同步的方法会比较快？

同步方法不会延迟执行，就算相对的异步方法执行很快。当 CPU 准备妥当时，同步方法会立即执行，会保证你只需要等待必要的 I/O 完成。但是，同步的方法在等待的运行期间会阻塞其他的事情发生。

在另一方面，异步的 `find` 方法会比较慢些，但它不会在碰到大的目录结构时执行失败（因为栈会逐步清除），并且 `find` 不会阻塞进程。

我们先来看看 `findSync` 的代码：

```
var fs = require('fs');  
var join = require('path').join;
```

```
exports.findSync = function (nameRe, startPath) {  
  var results = [];  
  
  function finder (path) {  
    var files = fs.readdirSync(path);  
  
    for (var i = 0; i < files.length; i++) {  
      var fpath = join(path, files[i]);
```

1

2

3

4

```

    var stats = fs.statSync(fpath);
    if (stats.isDirectory()) finder(fpath);
    if (stats.isFile() && nameRe.test(files[i])) results.push(fpath);
  }
}

finder(startPath);
return results;
}

```

- ❶ 接收一个正则用来从一开始的路径进行文件搜索。
- ❷ 存储匹配项的集合。
- ❸ 读取文件列表（包括目录）。
- ❹ 获取当前文件的路径。
- ❺ 获取当前文件的状态。
- ❻ 如果它是一个文件目录，继续使用新的路径调用 `finder`。
- ❼ 如果它是一个文件，并且满足搜索的匹配，将其添加到结果中。
- ❽ 初始化文件查找。
- ❾ 返回结果。

因为所有的方法都是同步的，我们能通过最后的 `return` 来得到结果，因为在所有的递归方法结束之前程序不会执行到那里。在 `try/catch` 中执行时，一旦遇到异常就会被抛出，并且被捕获，我们来看一个简单的例子。

```

var finder = require('./finder');
try {
  var results = finder.findSync(/file.*$/, '/path/to/root');
  console.log(results);
} catch (err) {
  console.error(err);
}

```

- ❶ 成功了，找到了文件列表。
- ❷ 噢，不！有异常情况，打印错误。

现在来看一下如何实现 `find` 方法并通过异步的方法处理这个问题。

```

var fs = require('fs');
var join = require('path').join;

```

```
exports.find = function (nameRe, startPath, cb) {  
  var results = [];  
  var asyncOps = 0;  
  var errored = false;  
  
  function error (err) {  
    if (!errored) cb(err);  
    errored = true;  
  }  
  
  function finder (path) {  
    asyncOps++;  
    fs.readdir(path, function (err, files) {  
      if (err) return error(err);  
  
      files.forEach(function (file) {  
        var fpath = join(path, file);  
  
        asyncOps++;  
        fs.stat(fpath, function (err, stats) {  
          if (err) return error(err);  
  
          if (stats.isDirectory()) finder(fpath);  
          if (stats.isFile() && nameRe.test(file)) results.push(fpath);  
  
          asyncOps--;  
          if (asyncOps == 0) cb(null, results);  
        })  
      })  
    })  
  
    asyncOps--;  
    if (asyncOps == 0) cb(null, results);  
  });  
  
  finder(startPath);  
}
```

- ❶ find 方法现在接受第三个参数作为一个回调。
- ❷ 为了知道是否已经完成遍历，需要一个计数器。
- ❸ 为了防止多个错误的调用，如果我们没有成功，我们会追踪什么时候发生了错误。
- ❹ 错误的处理句柄，用于确保如果有多个错误，回调只会执行一次。

- ⑤ 在每一个异步操作之前计数器自增。
- ⑥ 在这里需要一个闭包，这样我们才不会丢失文件引用。
- ⑦ 在每一个异步操作已经完成之后计数器自减。
- ⑧ 如果我们回到零了，那么已经完成并且没有错误，这时候可以在回调中返回结果。

我们不能像同步方法那样直接 `return` 结果，需要在执行结束后调用回调函数得到结果。为了知道何时结束，我们使用了一个计数器（`asyncOps`）。我们也需要意识到，每当有回调函数时，需要确保当异步调用完成时我们有一个闭包来使得我们需要的变量还保留着（这是我们使用一个 `forEach` 调用而不是标准的 `for` 循环的原因，关于这个更多请参考 <http://mng.bz/rqEA>）。

计数器（`asyncOps`）在执行一个异步操作（如 `fs.readdir` 或 `fs.stat`）之前增加，并且在异步操作的回调中减少。确切地说它在任何异步方法执行结束后减少（否则我们很快就会得到 0）。在一个成功的场景中，计数器 `asyncOps` 会在所有的递归异步操作完成后变成 0，并且可以把 `result` 传入回调函数中（`if (asyncOps == 0) cb(null, results)`）。在失败的场景中，`asyncOps` 永远不会变成 0，并且其中一个异常处理方法会被处罚并将错误通过回调传回。

在例子中，我们不能肯定 `fs.stat` 会是最后一个被调用的函数，因为我们可能有很多空目录，所以需要在每个地方都检查。还有一个简单的 `error` 封装来保证我们不会回调超过一次错误。假如你的异步操作和我们的例子一样返回了一个值或者一个 `error`，重要的一点是你必须确保只调用一次回调，因为这将导致很难排查的 `bug`。

计数器的替代方案

计数器并不是唯一可以跟踪一组异步操作是否完成的方案。取决于应用的需求，还可以递归地传入原来的回调。可以参照第三方模块 `mkdirp`（<https://github.com/substack/node-mkdirp>）。

现在有了一个异步的版本（`find`）并且可以通过标准的 Node 风格的回调方法来处理结果。

```
var finder = require('./finder');
finder.find(/file*/, '/path/to/root', function (err, results) {
  if (err) return console.error(err);
  console.log(results);
});
```

用第三方模块处理并行操作

并行操作会很难追踪，并且很容易出现 bug，所以需要使用第三方模块，例如 `async` (<https://github.com/caolan/async>)，来帮助你解决这个问题，另外一个替代方案是使用 `promises` 类库，例如 `Q` (<https://github.com/kriszov/q>)。

技巧 43 编写文件数据库

Node 的核心 `fs` 模块提供给你可以构建复杂的递归操作的工具，就像你在上一个技巧看到的。它还可以让你做其他复杂的任务，例如创建一个文件数据库。在这个技巧中，我们会写一个文件数据库来看 `fs` 模块中其他的特性，包括操作流。

问题

你需要一个简单并且快速的数据存储结构，并且保证一致性。

解决方案

通过追加日记使用内存数据库。

讨论

我们会写一个简单的 `Key/value` 数据库模块。这个数据库将为了速度而提供内存访问当前状态，并且使用追加日志的存储格式来持久化。使用追加日志的存储将提供给我们以下几个优势：

- 有效的 *I/O* 效率——我们只写到文件的最后。
- 持久——文件的上一个状态永远不会改变。
- 简单地创建备份——我们可以只复制该文件在任何时候获得数据库的在该点的状态。

文件中的每一行是一个记录。该记录是一个简单的 `JSON` 对象的两个适当关系，`key` 和 `value`。`key` 表示查找值的字符串。该 `value` 可以是任何 `JSON` 序列化，其中包括字符串和数字。让我们看看一些实例记录：

```
{"key": "a", "value": 23}
{"key": "b", "value": ["a", "list", "of", "things"]}
{"key": "c", "value": {"an": "object"}}
{"key": "d", "value": "a string"}
```

如果一个记录被更新，记录的新版本将在以后的文件中找到相同的键：

```
{"key": "d", "value": "an updated string"}
```

如果记录已被删除，它也会以 `null` 值追加到文件后：

```
{"key":"b","value":null}
```

当数据库被加载，该杂志将从上到下流中，建立在存储器中的数据库的当前状态。记住，数据不会被删除，所以它能够存储以下数据：

```
{"key":"c","value":"my first value"}
```

```
...
```

```
{"key":"c","value":null}
```

```
...
```

```
{"key":"c","value":{"my":"object"}}
```

在这种情况下，在某些时候，保存 `"my first value"` 作为键 `c`。后面，删除该键。然后，设置这个键是 `{"my":"object"}`。最新条目将在内存中装载，因为它代表了数据库的当前状态。

我们谈到的有关数据将被保存到文件系统。让我们来谈谈 API。

```
var Database = require('./database');  
var client = new Database('./test.db');
```

```
client.on('load', function () {  
  var foo = client.get('foo');  
  
  client.set('bar', 'my sweet value', function (err) {  
    if (err) return console.error(err);  
    console.log('write successful');  
  });  
});
```

```
client.del('baz');  
});
```

- ❶ 加载我们的数据库模块。
- ❷ 提供我们想要加载或者创建的数据库文件的路径。
- ❸ 当数据加载到内存中时，`load` 事件会被触发。
- ❹ 使用键 `foo` 来获取存储的值。
- ❺ 为键 `bar` 来设置一个值。
- ❻ 持久化到磁盘中可能会产生错误。
- ❼ 删除键 `baz`。写入后的回调参数是可选。

让我们深入到代码开始把这放在一起。我们将编写一个 `Database` 模块存储我们的逻辑。

它会继承 `EventEmitter`，所以我们可以发出事件传回给消费者（比如当数据库已加载所有数据，我们可以开始使用它）：

```
var fs = require('fs')
var EventEmitter = require('events').EventEmitter

var Database = function (path) {
  this.path = path

  this._records = Object.create(null)
  this._writeStream = fs.createWriteStream(this.path, {
    encoding: 'utf8',
    flags: 'a'
  })

  this._load()
}

Database.prototype = Object.create(EventEmitter.prototype)
```

❶

❷

❸

❹

❺

- ❶ 设置数据库存储的路径。
- ❷ 创建在内存中内部的所有记录的映射。
- ❸ 创建一个仅用于添加模式的写入流来处理磁盘写入。
- ❹ 加载数据库。
- ❺ 从 `EventEmitter` 继承而来。

我们要流存储的数据，并在完成后发出“load”事件。流将使我们能够处理数据，因为它是被读取的。流也是异步的，允许主机应用程序在加载数据做其他事情：

```
Database.prototype._load = function () {
  var stream = fs.createReadStream(this.path, { encoding: 'utf8' });
  var database = this;

  var data = '';
  stream.on('readable', function () {
    data += stream.read();
    var records = data.split('\n');
    data = records.pop();

    for (var i = 0; i < records.length; i++) {
      try {
        var record = JSON.parse(records[i]);
        if (record.value == null)
```

❶

❷

❸

❹

```

    delete database._records[record.key];
  else
    database._records[record.key] = record.value;
  } catch (e) {
    database.emit('error', 'found invalid record:', records[i]);
  }
}
});
stream.on('end', function () {
  database.emit('load');
});
}

```

- ❶ 读取可用的数据。
- ❷ 按行分割数据记录。
- ❸ 获取最后的一个可能未完成的记录。
- ❹ 如果记录有一个 `null` 值，删除该记录。
- ❺ 否则，对于所有的非 `null` 值，按照键来存储。
- ❻ 如果找到一个不可用的记录，触发一个错误事件。
- ❼ 当数据已经准备好可用时，触发一个 `load` 事件。

当我们阅读来自文件中的数据，发现所有存在的完整记录。

写入时保持结构来确保读取时的结构

当 `readable` 事件触发时，我们只是在最后 `pop()` 一下时，要对数据做什么？最后的记录通常都是一个空的字符串（''），因为我们每一行都以换行符（`\n`）结束。

一旦我们加载数据和发出 `load` 事件，客户端就可以开始进行与交互数据。让我们来看看这些方法，从最简单的 `get` 方法开始：

```

Database.prototype.get = function (key) {
  return this._records[key] || null;
}

```

- ❶ 返回键对应的值，如果没有，返回 `null`。

接下来我们看下 `updates`：

```

Database.prototype.set = function (key, value, cb) {
  var toWrite = JSON.stringify({ key: key, value: value }); + '\n'
}

```

```
if (value == null)
  delete this._records[key];
else
  this._records[key] = value;

this._writeStream.write(toWrite, cb);
}
```

❶ 把 JSON 对象转成字符串，然后添加一个新行。

❷ 如果是删除，把记录从内存中移除。

❸ 否则，按照键值在内存中设置。

❹ 把记录写到磁盘中，如果有回调则执行回调。

现在来添加一个语法糖来实现删除：

```
Database.prototype.del = function (key, cb) {
  return this.set(key, null, cb);
}
```

❶ 调用 set 方法把值设置位 null（存储一个删除的记录）。

这样我们就实现了一个简单的数据库模块。最后，需要导出构造函数：

```
module.exports = Database;
```

这里还有很多可以优化的地方，比如 flushing writes (<http://mng.bz/2g19>) 或者失败重试。对于功能更完备的基于 node 的数据库模块，可以参考 node-dirty (<https://github.com/felixge/node-dirty>) 或者 nstore (<https://github.com/creationix/nstore>)。

技巧 44 监视文件以及文件夹

以往需要处理一个文件时，客户端添加一个文件至一个目录（例如，通过 FTP）或者在文件被修改后重启 Web 服务器重启？你可以通过监听文件变化实现这两个需求。

Node 对于文件监听有两个实现。我们将在该技巧中谈谈何时使用这两个实现。但在核心，它们可以实现同样的事情：监听文件（目录）。

问题

你想要监听一个文件或目录并在文件更改时执行一个动作。

解决方案

使用 fs.watch 和 fs.watchFile。

讨论

在 Node 核心中很难看到针对同一个目的有多种实现。Node 文档建议尽可能使用 `fs.watch`，因为它被认为更可靠。但 `fs.watch` 不是跨平台的方法，而 `fs.watchFile` 是。为什么这么乱？

关于 `fs.watch`

为了在单线程的环境中实现异步 I/O，Node 的事件轮询渗入操作系统中。这也提供了一个性能优势，作为操作系统能够让进程立即知道什么时候一些新的 I/O 模块是随时可以办理的。操作系统有很多方法通知进程，这也是为什么我们使用 `libuv`。实现文件监听的核心方法就是 `fs.watch`。

`fs.watch` 结合所有这些不同类型的事件系统成具有共同的 API 一体的方法来提供以下几个特性：

- 一个更可靠的实现使得文件改变的事件能够总是被执行。
- 一个更快的实现，当事件发生时能够立即通知到 Node 进程。

下面让我们看一下老方法。

关于 `fs.watchFile`

还有一个，老的实现文件监听的方法叫作 `fs.watchFile`。它没有渗入到通知系统，而是在一个时间段内不停地轮询来看是否文件有变化。

`fs.watchFile` 并不成熟也不快，但这个方法的优势是跨平台的，并且在网络文件系统（例如 SMB 和 NFS）上更可靠。

我该用哪个

建议用 `fs.watch`，因为它并不跨平台，最好有一个测试用例来测试它是不是你想要的。

让我们写一个程序来玩转一下文件监控以及看一下它所提供的 API。首先，创建一个叫 `watcher.js` 的文件：

```
var fs = require('fs');
fs.watch('./watchdir', console.log);
fs.watchFile('./watchdir', console.log);
```

在与 `watcher.js` 文件相同路径下创建一个叫 `watchdir` 的文件夹：

```
mkdir watchdir
```

然后，打开几个终端。在第一个终端中运行：

```
node watcher
```

然后在第二个终端，切换到 `watchdir` 目录下：

```
cd watchdir
```

在两个终端下，最好是同时打开，我们将在 `watchdir` 目录下做些变化，然后看 Node 对这些变化的反应。我们先创建一个新的文件。

```
touch file.js
```

我们可以看到 Node 的输出：

```
rename file.js  
change file.js  
{ dev: 64512,  
  mode: 16893,  
  nlink: 2,  
  ... } { dev: 64512,  
  mode: 16893,  
  nlink: 2,  
  ... }
```

①

②

① 一对 `fs.watch` 事件很快就到来。这里 `fs.watch` 只触发两个事件。第二个参数，`file.js` 是接收事件的文件。

② `fs.watchFile` 事件迟一点到来，并且有着不一样的响应。它包含了两个 `fs.Stats` 的对象，来表示文件当前和过去的状态。在这里它们是一样的，因为文件刚刚创建。

好的，现在我们已经创建了一个文件，让我们用相同的命令更新它的更改时间：

```
touch file.js
```

现在再看一下输出，我们看到只有 `fs.watch` 对这个更改有响应：

```
change file.js
```

因此，如果监听通过 `touch` 来更新文件对你的应用很重要，那么使用 `fs.watch`。

fs.watchFile 以及目录

在监听一个目录时，许多对文件的更新不会被 `fs.watchFile` 监听到。如果你想使用 `fs.watchFile`，那么可以监听单个文件。

我们尝试移动文件：

```
mv file.js moved.js
```

在终端，我们看到下面的输出，表示两个接口都监听到了变化：

```
rename file.js
rename moved.js
{ dev: 64512,
  mode: 16893,
  nlink: 2,
  ... } { dev: 64512,
  mode: 16893,
  nlink: 2,
  ... }
```

❶ `fs.watch` 从旧的名字改成新的，报告两次 `rename` 事件。

❷ `fs.watchFile` 表示文件已经被修改。

这里的要点是测试使用你想使用的 API。但愿，这个 API 在未来更加稳定。阅读文档，以获得最新的进展（http://nodejs.org/api/fs.html#fs_watch_filename_options_listener）。下面是一些提示：

- 运行测试，优先考虑 `fs.watch`。事件是不是按你的预期触发？
- 如果你打算看一个单一的文件，不要看它在目录中；否则你会收到很多事件触发。
- 如果在变化中比较文件是重要的，`fs.watchFile` 提供了开箱即用的方法。否则，你需要手动使用 `fs.watch` 管理统计。
- 仅仅因为你的 Mac 上 `fs.watch` 能够正常运行不意味着它同样就能在 Linux 服务器上执行。确保开发环境和生产环境都被测试。

尝试一下，然后明智地观察结果。

6.2 总结

在本章中，我们谈到通过一些使用 `fs` 模块的技术。在寻找配置文件装载和递归文件处理时，介绍了异步和同步使用。我们也看了文件描述符和文件锁定。最后，我们实现了一个文件数据库。

希望这已扩大你的一些使用 `fs` 模块可能的概念的理解。这里有几个小贴士：

- 同步方法比异步方法更简单，更好，但要注意性能问题，尤其是当你在写服务端代码时。
- 咨询文件锁定是一个有用的机制，只要所有进程在多个进程共享资源时遵循相同的约定。

- 某种反应完成后需要进行跟踪并行异步操作。虽然它有助于了解如何使用计数器或递归方法，但也可以考虑使用像 `async` 那样屡试不爽的第三方模块。
- 如何使用一个普通的文件，将决定你遵循哪种行动的方式。如果它是一个大的文件或数据块，可以考虑使用流的方法。如果它是一个更小的文件或直到你整个加载后才能使用的文件，考虑批量方法。如果想改变一个特定文件的一部分，你可能想坚持使用 POSIX 文件的方法。

在下一章，我们将看看 Node I/O 中的另一主要内容：网络。

7 网络：Node 真正的 “Hello, World”

本章概要

- 网络的概念和与 Node 的关系
- TCP、UDP、HTTP 客户端和服务端
- DNS
- 网络加密

Node.js 平台自己的卖点就是开发快速稳定的网络应用。写网络软件，你需要理解网络原理和连接协议。在下一节的课程中，我们解释网络如何围绕清晰的技术栈设计。此外，Node 如何实现这些协议和它们的 API。

在本章你将会学到 Node 的网络模块如何工作。这包括 `dgram`、`dns`、`http` 和 `net` 几个模块。如果你不确定一些网络术语，诸如 *socket*、*packet* 和 *protocol*，那么不必担心：我们也会介绍关键的网络概念，让你为网络编程打好坚实的基础。

7.1 Node 中的网络

本节是对网络的介绍。你将会学到网络 *layers*、*packets*、*sockets*——是这些构成了网络。这些概念都是了解 Node 网络 API 的关键。

7.1.1 网络技术

网络技术能够迅速变得势不可当。为了集中在同一页上，我们提供了表 7.1，其中总结了构成这一章的基础的主要概念。

为了解 Node 的网络 API，关键是学习 layers、packets、sockets 和所有其他构成网络的部分。如果你不了解 TCP（传输控制协议）和 UDP（用户数据报协议），那么何时使用这些协议会变得很困难。在本节中，我们介绍你需要知道的、探索的概念有点儿多，因此通过本节你将会有一个坚实的基础。

表 7.1 网络概念

项	描 述
Layer	代表一个逻辑组的网络协议切片。我们工作在应用层，是顶层；物理层是底层
HTTP	超文本传输协议——一个基于 TCP 的应用层客户端-服务端协议
TCP	传输控制协议——允许客户端到服务端的双向通信，而且目的是创建像 HTTP 一样的应用层协议
UDP	用户数据报协议——一个轻量协议，一般在期望速度而非可靠性时选择它
Socket	一个 IP 和一个端口号的结合通常被成为一个 socket
Packet	TCP 数据包也被看作——一个数据块和一个首部
Datagram	UDP 相当于一个包
MTU	最大传输单元——一个协议数据单元的最大尺寸。每一层都有一个 MTU：IPv4 的至少是 68 个字节，而以太网 v2 是 1500 字节

如果你是负责实施 HTTP 或即使使用 UDP 在低延迟游戏代码之上运行高层协议，那么你应该了解每一个概念。我们在接下来的几节内容中更详细地分解每一个概念。

Layers

构成网络和网络技术的协议栈和标准可以被抽象为一个层。底层表示物理媒体——以太网、蓝牙、光纤——这是一个插口、电压和网络适配器的世界。作为软件开发者，我们在硬件层的上层工作。当谈论到 Node 网络，我们关心的是互联网协议（IP）套件的应用的传输层。

层是最好的视觉表达。图 7.1 描述了数据包的逻辑网络层。较底层的物理和数据链路层协议包包裹的更高层的协议。

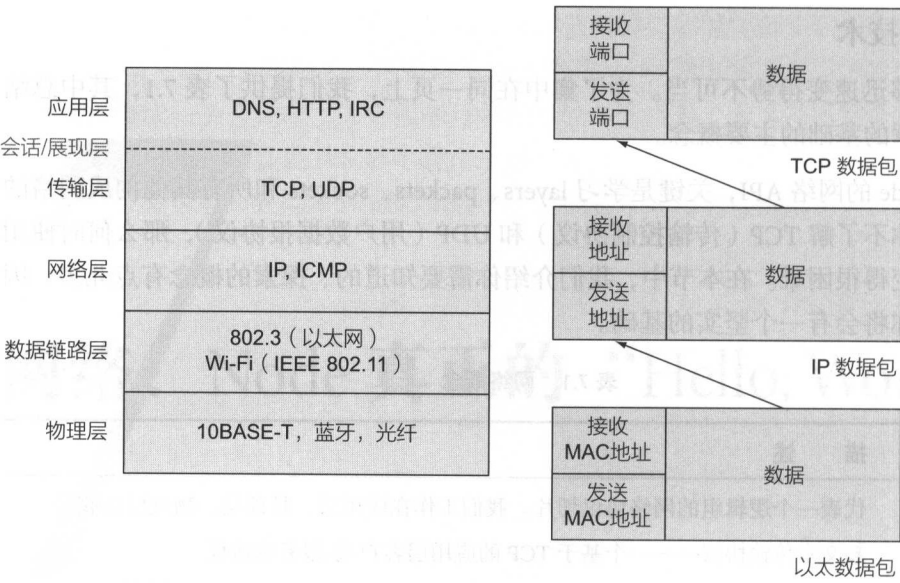


图 7.1 协议被分为七个逻辑分层。连续的分层协议对数据包进行封装

数据包被连续的协议层包裹的。一个 TCP 包，可能代表了一系列从 HTTP 请求报文的一部分，包含在一个 IP 包，这反过来又包裹通过以太网分组的数据部分。回到图 7.1，从通过传输层和应用层切 HTTP 请求的 TCP 数据包：TCP 是传输层，用于创建更高级别的 HTTP 协议。其他层也参与，但我们并不总是知道哪些特定的协议用于每一层：HTTP 始终传输 TCP/IP，但除此之外，Wi-Fi 或以太网可以使用，你的程序将不会知道的差别。

图 7.2 显示了网络层是怎么被每个协议层包裹的。注意，数据是从来不可以跨层的，我们不讨论传输层协议与网络层交互的多个步骤。

写 Node 程序，你应该明白，HTTP 使用 TCP 实现，因为节点的 http 模块是建立在网络上的模块中发现的底层 TCP 实现的。但你并不需要了解以太网、10BASE-T 或蓝牙工作原理。

TCP/IP

你可能听说过 TCP/IP——这就是被我们称为互联网协议套件的东西，因为传输控制协议（TCP）和互联网协议（IP）是最重要而且是最早被这一标准定义的。

在互联网协议中，一个主机通过一个 IP 地址识别。在 IPv4 中，地址是 32 位的，是受到地址可用空间限制的。过去十年中，IP 一直受到争议，因为地址即将耗尽。为了解决这个问题，被称为 IPv6 的协议被开发出来。

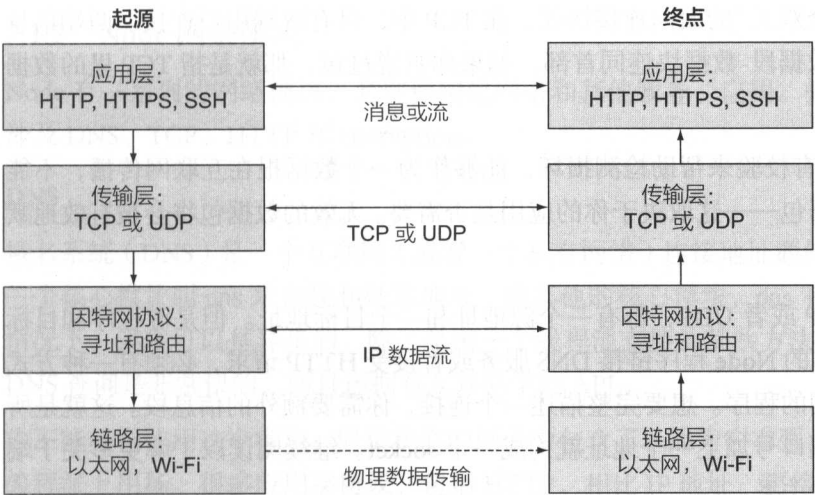


图 7.2 网络层封装

你能够使用 `net` 模块创建 TCP 连接。这允许你实现核心模块不支持的应用层协议：IRC、POP，甚至 FTP 能够使用 Node 核心模块实现。如果你发现自己需要访问非标准 TCP 协议，或许是使用在你公司的什么东西，那么 `net.Socket` 和 `net.createConnection` 会帮你轻松解决。

Node 在几方面同时支持 IPv4 和 IPv6：`dns` 模块能够查询 IPv4 和 IPv6 记录，而且网络模块能够在 IPv4 和 IPv6 网络从主机发送和接收数据。

有趣的是它并不能保证数据完整和传输。为了可靠的通信，我们需要 TCP 一样的传输层。很多时候并不需要传输，虽然这样但也需要一个轻量的协议，这就需要 UDP 了。下一节详细讨论 TCP 和 UDP。

UDP 与 TCP 的差异

数据报是 UDP 通信的基础单元。这些消息是自包含的，包括源、目标和一些用户数据。UDP 不能保证传输和消息顺序，也不会提供保护来防止重复数据。你使用的 Node 程序中的很多协议都基于 TCP，但有时 UDP 也有用。如果传输不是首要的，但是很需要性能，那么 UDP 就是更好的选择。一个例子就是视频流服务，在面对更大的吞吐量时，偶然的毛刺是可以折中考虑的。

TCP 和 UDP 都使用了相同的网络层——IP。都是为应用层提供服务的。但它们非常不同。TCP 面向连接且可靠的字节流服务，UDP 则是基于数据报的，而且不能保证数据传输。

与此不同，TCP 是全双工¹的面向连接协议。在 TCP 中，只有两端用户连接。两端信息传输的基本单元是数据段-数据块连同首部。如果你听说过包，那就是指 TCP 里的数据段了。

虽然 UDP 数据包含有校验来帮助检测损坏，能够作为一个数据报在互联网传播，不能自动重传损坏的数据包——这取决于你的应用是否需要。无效的数据包将会被有效地默默丢弃。

每个包，无论是 TCP 或者 UDP，都有一个源地址和一个目标地址。但是源程序和目标程序都很重要。当你的 Node 程序链接 DNS 服务或者接受 HTTP 请求，必须有一种方式来映射包和生成它们的程序。想要完整描述一个连接，你需要额外的信息段。这就是所说的端口——一个端口号加上一个地址就构成一个 socket。继续阅读以了解更多关于端口和它们与 socket 的关系。

Sockets

网络的基础单元，从程序员的角度看是 socket。一个 socket 是一个 IP 地址和一个端口号的结合体——TCP 和 UDP 都有 sockets。就像你在前面章节看到的，TCP 连接是全双工的，打开一个连接指定 host 并允许接收和发送。即使长连接是正确的，历史上“socket”意味着伯克利 Sockets 接口。

Berkeley Sockets API

Berkeley Sockets，发布于 1983 年，是用于网络 sockets 的 API。这是 TCP/IP 套件的起源。即使起源是在 UNIX 上，微软的 Windows 包含遵从贴近 Berkeley Sockets 的网络栈。

标准的 TCP/IP 服务的端口号众所周知。它们包括 DNS、HTTP、SSH，以及更多。由于历史原因，这些端口号通常是奇数。TCP 和 UDP 端口是不同的，这样它们可以叠加。如果一个应用层协议同时依赖 TCP 和 UDP 连接，那么通常做法是两个连接使用相同的端口号。同时使用 UDP 和 TCP 协议的例子是 DNS。

在 Node 中，你能够使用 net 模块创建 TCP sockets，UDP 是使用 dgram 模块。其他网络协议也是支持的——DNS 是个好例子。

下一节看一下 Node 核心模块包含的应用层协议。

¹全双工：消息会在同一个连接中发送和接收。

7.1.2 Node 网络模块

Node 有一系列的网络模块，允许你构建网络和其他服务、应用。接下来的几节我们会涉及 DNS、TCP、HTTP 和 encryption。

DNS

域名系统（DNS）是一个互联网（或者一个私有网络）连接地址源的命名系统。Node 有一个核心模块叫 `dns` 来查找和处理地址。像其他的核心模块，`dns` 有异步 APIs。在这种情况下，实现可以是异步的，除了是由一个线程池支持的某些方法。这意味着 Node 的 DNS 查询是非常快的，而且还拥有友好易学的 API。

你不能经常使用这个模块，但是我们的例子中包含了，因为它有强大的 API 能够在网络编程派上用场。很多应用层协议，包括 HTTP，相比 IP 地址，更接受主机名。

Node 也提供一些我们更加熟悉的网络协议模块，例如 HTTP。

HTTP

HTTP 对绝大多数开发者来说很重要。无论你是构建网络应用还是调用网络服务，你或许会对 HTTP 感兴趣。Node 的 `http` 模块基于 `net`、`stream`、`buffer` 和 `events` 模块。这是底层的，但是我们能够使用它轻松创建 HTTP 服务端和客户端。

由于 Web 对开发者的重要性，我们包含一些练习来探索 Node 的 `http` 模块。同样，当我们使用 HTTP 的时候通常会用 `encryption`——Node 通过 `crypto` 和 `tls` 模块支持加密。

Encryption

你应该知道 SSL——Secure 连接层，因为它是浏览器的安全浏览方式。不仅仅是 HTTP 传输需要加密，其他服务，比如 email，加密消息也需要。使用 TLS 加密 TCP 连接：传输层安全。Node 的 `tls` 模块使用 OpenSSL 实现。

这个加密类型被称作公钥加密。客户端和服务端必须有一个私钥。服务端就能够通过其公钥以实现客户端加密。为了加密这些消息，通过服务端的私钥是必需的。

Node 通过创建 TCP 服务器支持多种加密算法来支持 TLS。TCP 服务继承自 `net.Server`——一旦你获得了 Node 中 TCP 客户端和服务端的头，加密连接仅仅是这些原则的扩展而已。

如果要部署 Web 应用程序，对 Node TLS 的扎实理解是非常重要的。人们越来越关注安全和隐私，不幸的是 SSL/TLS 的设计方式，会让程序员的错误使用导致安全漏洞。

在 Node 网络的最后一方面，我们会为本章介绍：Node 如何能够在系统阻塞时通过异步 APIs 使用网络技术。

处理 I/O 操作，如果你想真正了解 Node 的工作原理，理解非阻塞 I/O、线程池和异步 APIs 的区别是很重要的。

对于那些有兴趣阅读更多关于 libuv 和网络的内容，免费提供的图书，*An introduction to libuv* (<http://nikhilm.github.io/uvbook/networking.html#tcp>) 有针对网络部分的内容，如 TCP、DNS 和 UDP。

现在开始网络技术：TCP 客户端和服务端。

7.2 TCP 客户端和服务端

Node 拥有简单的 API 来创建 TCP 链接和服务。在 `net` 模块中能找到大部分底层的类和方法。在下一个技巧，你将会学习如何创建一个 TCP 服务，跟踪与它连接的客户端。很酷的是更高层的协议，比如 HTTP，是基于 TCP API 的，因此一旦你投入 TCP 客户端和服务端的怀抱，你就能够真正地开始开发一些更多的特性。

技巧 45 创建 TCP 服务端和客户端

`net` 模块构成了 Node 网络特性的基础。这节展示了如何创建 TCP 服务。

问题

你想要启动自己的 TCP 服务，绑定一个端口，通过网络发送数据。

解决方案

使用 `net.createServer` 创建一个服务，然后调用 `server.listen` 绑定到一个端口。连接服务端，或者用命令行工具 `telnet` 或者创建一个进程内的客户端连接副本，`net.connect`。

讨论

`net.createServer` 方法返回一个对象，能够用来监听一个指定的 TCP 端口应对连接。当一个客户端创建了一个新连接，传递给 `net.createServer` 回调函数将会执行。回调接受一个面向事件的连接对象。

这个服务对象是 `net.Server` 的一个实例，仅仅是对 `net.Socket` 类的一个封装。有趣的是需要注意 `net.Socket` 是使用一个双工流来实现的——了解更多流，参见第 5 章。

在进入更多原理之前，我们来看一个你能够运行 `telnet` 连接的例子。下面展示了一个简单的 TCP 服务端，接收连接并且返回数据给客户端。

例子 7.1 一个简单的 TCP 服务器

```
var net = require('net');  
var clients = 0;  
  
var server = net.createServer(function(client) {  
  clients++;  
  var clientId = clients;  
  console.log('Client connected:', clientId);  
  
  client.on('end', function() {  
    console.log('Client disconnected:', clientId);  
  });  
  
  client.write('Welcome client: ' + clientId + 'rn');  
  client.pipe(client);  
});  
  
server.listen(8000, function() {  
  console.log('Server started on port 8000');  
});
```

- ❶ 加载网络模块。
- ❷ 创建 ID 来引用连接的每一个客户端。
- ❸ 当客户端连接时，ID 自增，并且存放在当前局部作用域下。
- ❹ 绑定 end 事件来追踪客户端断开连接。
- ❺ 使用客户端的 ID 给每个客户端打个招呼。
- ❻ 使用管道把客户端的数据返回给客户端。
- ❼ 绑定到 8000 端口开始接收新链接

我们尝试这个例子，运行 `node server.js` 来启动服务，然后运行 `telnet localhost 8000` 来连接。你能够链接数次来查看 ID 自增。如果你断开连接，含有正确客户端 ID 的消息将会被打印。

很多使用 TCP 客户端和服务端的程序加载 `net` 模块❶。一旦它被载入，TCP 服务端能够使用 `net.createServer` 创建，这实际上只是 `new net.Server` 和一个事件侦听器 `listener` 的快捷方式。一个服务被实例化之后，它能够通过 `server.listen` ❼监听连接端口。

回显客户端的数据，可以使用管道❺。Sockets 是流，所以你能够使用第 5 章看到的标准流 API 方法。

在本例中，我们使用数字 ID ②来跟踪每个客户端③。连接的客户端的总量是通过创建名为 `clientId` 的连接回调的变量存储在回调的作用域里。

每当客户端连接⑤或断开④这个值被显示。传递到服务器的回调客户端参数实际上是一个 `Socket`——你可以使用 `client.write` 写入，而且数据将通过网络发送。

需要注意的重要一点是，添加到 `socket` 的任何事件监听器回调将共享相同的作用域——它会创建回调内的作用域。这意味着客户端 ID 对于每个连接是唯一的，你还可以存储，客户端可能需要其他值。这形成了 Node 客户端-服务器应用程序使用的通用模式。

接下来的技巧的例子是：同一进程中添加客户端连接。

技巧 46 使用客户端测试 TCP 服务端

Node 能够轻松地在同一个进程创建 TCP 服务端和客户端，这是一个测试网络应用程序特别有用的方法。在这个技巧中，你会学习如何创建 TCP 客户端，以及使用它测试服务端。

问题

你想要测试 TCP 服务器。

解决方案

使用 `net.connect` 连接服务端端口。

讨论

由于 TCP 和 UDP 的端口工作原理，完全有可能在一个进程中创建多服务端和客户端。举个例子，一个 Node 的 HTTP 服务也能够运行一个简单的 TCP 服务在另一个端口允许远程管理连接。

在技巧 45 中，我们展示了一个 TCP 服务能够通过客户端发出的唯一 ID 跟踪客户端。下面写一个测试来确定这一点。

例子 7.2 展示了如何创建一个进程内服务的客户端连接，然后在数据通过网络发送的时候运行断言。当然，技巧并没有运行在真实的网络中，因为所有的内容都发生在同一个进程中，但是它能够很轻易地适应那种方式；只是拷贝程序到服务端并且在客户端指定 IP 地址和主机名。

例子 7.2 创建 TCP 客户端来测试服务器

```
var assert = require('assert');
var net = require('net');
```

```
var clients = 0;
var expectedAssertions = 2;

var server = net.createServer(function(client) {
  clients++;
  var clientId = clients;
  console.log('Client connected:', clientId);

  client.on('end', function() {
    console.log('Client disconnected:', clientId);
  });

  client.write('Welcome client: ' + clientId + '\r\n');
  client.pipe(client);
});

server.listen(8000, function() {
  console.log('Server started on port 8000');

  runTest(1, function() {
    runTest(2, function() {
      console.log('Tests finished');
      assert.equal(0, expectedAssertions);
      server.close();
    });
  });
});

function runTest(expectedId, done) {
  var client = net.connect(8000);

  client.on('data', function(data) {
    var expected = 'Welcome client: ' + expectedId + '\r\n';
    assert.equal(data.toString(), expected);
    expectedAssertions--;
    client.end();
  });

  client.on('end', done);
}
```

❶

❷

❸

❹

❺

❻

❼

❽

❶ runTest 函数接收一个回调，这样额外的测试可以更好地进行安排。

❷ 在测试完成之后，会检查计数器来确认测试是否已经执行。

- ❸ 测试和断言运行一次后，服务器便可以被关闭。
- ❹ `runTest` 函数连接服务器，确认展示所期望的用户 ID，然后断开连接。
- ❺ `net.connect` 用于连接服务器，它返回一个 `EventEmitter` 对象可以用于监听各种事件。
- ❻ 客户端的 `data` 事件是用来获取服务器在客户端连接后展示的消息。
- ❼ 当数据发送后，断开客户端连接。
- ❽ 当客户端接收完数据后，运行回调函数。

这是一个长例子，但是它都是围绕一个相对简单的方法：`net.connect`。这个方法接受一些可选参数描述远程主机。这里我们已经指定了一个端口号，但是第二个参数可以是一个主机名或者 IP 地址——默认是 `localhost` ❺。它也能够接受一个回调，被用来为已经连接的其他端写数据。记住，TCP 服务器是全双工的，所以两端都能够接收和发送数据。

一旦服务器开始监听，例子中的 `runTest` 方法就会运行 ❶。它接受一个预期的客户端 ID 和一个叫 `done` 的回调 ❷。客户端被连接回调就会被触发，通过订阅的 `data` 事件接收一些数据 ❸。

当客户端断开连接的时候，`end` 事件将会被触发。我们为它绑定了 `done` 事件 ❹。当 `data` 数据回调中的 `test` 结束时，我们调用 `client.end` 断开 `socket` 连接，但是 `end` 事件也会在服务器关闭连接时触发。

`data` 事件是主要测试执行的地方 ❷。期望的信息通过事件中的 `data` 被传进 `assert.equal`。`data` 是一个 `buffer`，因此 `toString` 需要在断言时候调用。一旦测试结束，`end` 事件已经被触发 ❹，传递给 `runTest` 的回调将会被执行。

在这里我们使用了 `runTest` 的两次调用，一次在回调内。一旦两次都执行，期望的断言值就会被检查 ❺，服务就会停止 ❻。

错误句柄

如果你需要收集 TCP 连接所产生的错误，只需要对 `net.connect` 返回的 `EventEmitter` 对象监听 `error` 事件即可。如果不这样，就会产生异常；这是 Node 的标准行为。

不幸的是，处理一系列的网络连接是不容易的。在这样的情况下，更好的技术是使用 `domain` 模块。使用 `main.create()` 创建一个新的 `domain` 会导致 `error` 事件发送至 `domain`；然后你能够通过订阅域的 `error` 事件来处理它们。

关于 `domains` 更多的信息，参考技巧 21。

这个例子强调了两个重要的东西：客户端和服务端能够同时运行在进程中，而且 Node TCP 客户端和服务端都容易单元测试。如果例子中的服务端是我们无权控制的远程服务，那么我们就可以创建一个“mock”服务，来明确表达要测试的客户端代码。这构成了大多数开发人员编写测试与编写 Node 应用的基础。

下一个技巧我们将会通过查看和研究 Nagle 算法如何改善网络拥塞来深挖 TCP 网络。

技巧 47 改进实时性低的应用

虽然 Node 的 net 模块是相对高层的，但它为一些底层的功能提供访问方法。一个例子就是判别是否使用 Nagle 算法的 TCP_NODELAY 的标志控制。本技巧解释了 Nagle 算法是什么，什么时候应该使用，如何在具体的 socket 中关闭它。

问题

要提高一个实时应用的连接等待时间。

解决方案

使用 `socket.setNoDelay()` 开启 TCP_NODELAY。

讨论

有时候集中起来处理要比分开好。每天都有数以百万计的产品被运至世界各地，但它们没有在同一时间承载同一个——相反它们根据最终的目的被分组。TCP 用同一种方式工作，而且这个特性是 Nagle 算法实现的。

Nagle 算法描述是当一个连接有未确认的数据，小片段应该保留。当足够的数据已被收件人确认，这些小片段将被分批成能够被传输的更大的片段。

在很多小的数据包传输的网络，理想的情况将小的包集合起来一起发送以减少拥堵。但有时等待时间比其他都重要，所以传送小包是非常重要的。

这对互动应用尤其重要，像 ssh 或者 X Window 系统。在这些应用中，体积小的消息应毫不延迟地输送到达实时反馈的感觉。图 7.4 展示了这个概念。

Node 中的某些类通过关闭 Nagle 算法获得更好的效率。举个例子，你可能已经创建了一个 REPL，当用户输入消息时传输单个字符，或者一个游戏来传输玩家的本地数据。下一节展示了一个程序关闭了 Nagle 算法的效果。

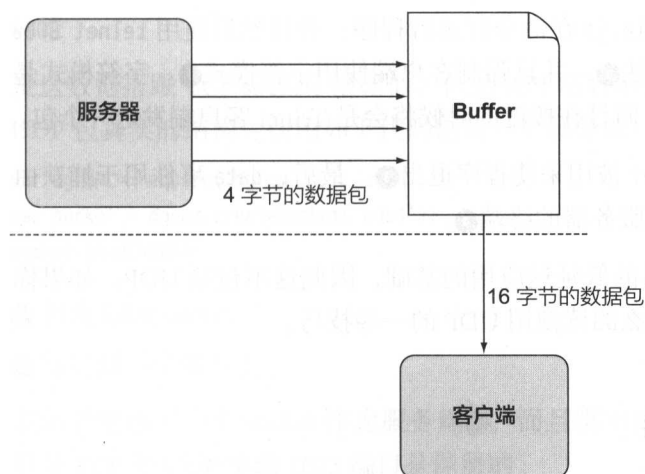


图 7.4 当使用 Nagle 算法时，这一层负荷时会收集更小的数据包

例子 7.3 关闭 Nagle 算法

```
var net = require('net');
var server = net.createServer(function(c) {
  c.setNoDelay(true);
  c.write('377375042377373001', 'binary');
  console.log('server connected');
  c.on('end', function() {
    console.log('server disconnected');
    server.unref();
  });
  c.on('data', function(data) {
    process.stdout.write(data.toString());
    c.write(data.toString());
  });
});
server.listen(8000, function() {
  console.log('server bound');
});
```

- ❶ 关闭 Nagle 算法。
- ❷ 强制客户端使用字符模式。
- ❸ 调用 `unref()` 来确保最后一个客户端断开后，退出程序。
- ❹ 把从客户端获取到的字符打印到服务器的终端上。

为了运行这个例子，通过 `node nagle.js` 在命令行运行程序，并且然后使用 `telnet 8000` 获得连接。服务端关闭了 Nagle 算法❶，并且强制客户端使用字符模式❷。字符模式是 Telnet 协议（RFC 854）的一部分，而且在按键的时候将会是 Telnet 客户端发送一个包。

然后当没有其他客户端连接时 `unref` 被用来使程序退出❸。最后，`data` 事件用于捕获由客户端发送的字符，并将其打印到服务器的终端❹。

这种技术作为创建数据完整性很高的低延迟应用的基础，因此这不包括 UDP。如果你真的想更多地控制数据的传输，那么阅读使用 UDP 的一些技巧。

7.3 UDP 客户端和服务端

对比 TCP，UDP 是更简单的协议。对你来说意味着更多的工作：不仅仅是响应数据的发送和接收，你不得不准备迎接 UDP 易变的特性。UDP 适合查询和响应的协议，这就是为什么会用在域名系统（DNS）上。它也是无状态的。如果你想要传输数据并且对数据完整性要求不高，那么 UDP 是个好选择。这听起来不太寻常，但是有些应用适合这些特性——流媒体协议和在线游戏通常使用 UDP。

如果想要构建一个视频流服务，你能够通过 TCP 传输视频，但是每个包都会有很大的开销以确保传输。使用 UDP，没有简单的方式发现数据丢失，但是视频不需要关心偶尔的毛刺——你只需要数据尽量快。事实上，一些视频和图像格式能够允许少量的数据丢失：JPEG 格式能在一定程度上抗损。

下一个技巧结合 Node 文件系统和 UDP 来创建一个简单的服务器来传输文件。虽然这可能会导致数据丢失，但是当你关心传输速度时也可能是有用的。

技巧 48 通过 UDP 传输文件

本技巧实际上是通过一个流发送数据到一个 UDP 服务器，而不是创建一个通用的文件传输机制。你能够使用它学习 Node 基础的数据报 API。

问题

你想要使用数据报从一个客户往服务端传输数据。

解决方案

使用 `dgram` 模块创建数据报 sockets，然后使用 `socket.send` 发送数据。

讨论

发送数据报跟 TCP sockets 很相似，但是 API 略有不同，而且数据报有自己的规则映射 UDP 包真实的结构。使用如下代码创建一个服务端：

```
var dgram = require('dgram');  
var socket = dgram.createSocket('udp4');  
socket.bind(4000);
```

❶

❷

❶ 创建 UDP socket。

❷ 绑定到一个端口上。

本例子创建了一个 socket 作为服务端❶，而且绑定到一个端口❷。端口可以是任意的，但是 TCP 和 UDP 中前 1023 端口是保留的。

客户端 API 与 TCP sockets 是不同的，因为 UDP 是无状态协议。你必须一次性写一个数据包，而且数据包（数据报）必须相对小——小于 65507 字节。数据报最大大小依赖网络 Maximum Transmission Unit (MTU)。64KB 是上限，但通常不使用，因为大的数据包可以被网络丢弃。

使用 `dgram.createSocket` 创建一个客户端 socket 与服务端相同。发送一个数据报需要一个 buffer 来承载，用偏移量来表明 buffer 中消息的开始、消息的长度、服务端口、远程 IP 和一个可选的回调，当消息发出时会被触发：

```
var message = 'Sample message';  
socket.send(new Buffer(message), 0, message.length, port, remoteIP);
```

例子 7.4 中在一个程序里包括了一个客户端和一个服务端。为了运行它，你必须发表两个命令：`node udp-client-server.js server` 来运行服务端，然后是 `node udp-client-server.js client remoteIP` 启动客户端。如果你在本地运行，`remoteIP` 选项可以省略；我们设计了这个例子在一个单独文件中，因此你能够轻松地拷贝它到另一台电脑测试网络发送文件或者本机网络。

例子 7.4 UDP 客户端和服务端

```
var dgram = require('dgram');  
var fs = require('fs');  
var port = 41230;  
var defaultSize = 16;  
  
function Client(remoteIP) {  
  var inStream = fs.createReadStream(__filename);
```

❶

```
var socket = dgram.createSocket('udp4');2

inStream.on('readable', function() {
  sendData();3
});

function sendData() {
  var message = inStream.read(defaultSize);4

  if (!message) {5
    return socket.unref();
  }

  socket.send(message, 0, message.length, port, remoteIP,
    function(err, bytes) {6
      sendData();
    }
  );
}

function Server() {
  var socket = dgram.createSocket('udp4');7

  socket.on('message', function(msg, rinfo) {8
    process.stdout.write(msg.toString());
  });

  socket.on('listening', function() {9
    console.log('Server ready:', socket.address());
  });

  socket.bind(port);
}

if (process.argv[2] === 'client') {10
  new Client(process.argv[3]);11
} else {
  new Server();
}
```

❶ 从当前的文件创建一个可读流。

❷ 创建一个新的数据流 socket 来作为客户端使用。

- ③ 当可读流准备好后，开始发送数据到服务器上。
- ④ 使用 `stream.read(size)` 来读取数据块。
- ⑤ 客户端完成任务后，当不需要时调用 `unref` 来安全地关闭它。
- ⑥ 否则，发送数据到服务器。
- ⑦ 创建一个 `socket` 来提供服务。
- ⑧ 当 `message` 事件被触发时，打印数据到终端上。
- ⑨ 表示服务器已经准备，可以提供给客户端连接来打印消息。
- ⑩ 检查命令行选项来确定是运行客户端还是服务端。
- ⑪ 接受其他可选的配置来连接远程的 IP 地址。

当你运行这个例子，它开始检查命令行选项来查看是否需要客户端或者服务端⑩。它也能够接受一个选项参数，因此你能够连接远程服务器⑪。

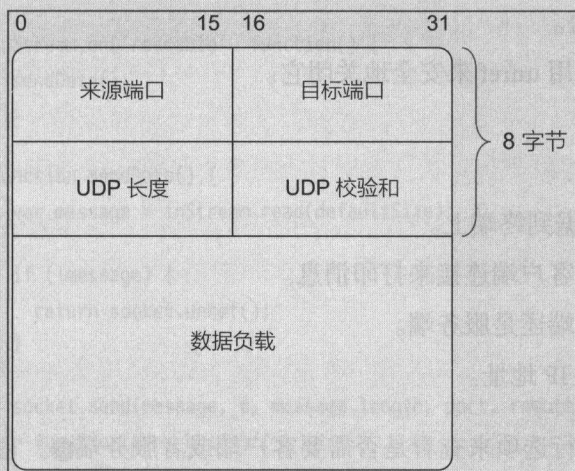
如果客户端被指定，那么一个新的客户端将会随着一个新的数据报 `socket` 创建②。这里涉及通过 `fs` 模块使用一个可读流，所以我们有一些数据发送到服务端①——我们使用了 `__filename` 使它能读取到当前文件，但是你能够用它发送任意文件。

在发送数据之前，我们需要确认文件是打开的，而且准备好被读取，因此 `readable` 事件被订阅了③。这个事件的回调执行了 `sendData` 方法。它会在每个文件片段都重复调用——使用 `Stream.read` ④一次会读取一小片段文件，因此 UDP 包如果太大就允许默默丢弃。`socket.send` 方法用来向服务端发送数据⑥。读取文件时返回的 `message` 对象是一个 `Buffer` 实例，能够直接传给 `socket.send`。

当所有的数据已经被读取，最后的片段被置空。当 `socket` 不再需要，`socket.unref` ⑤方法会被调用使程序退出。在这种情况下，一旦发送了最后的消息就会如此。

数据报构成和数据报大小

UDP 相对简单。它们由源端口、目标端口、数据报长度、校验值和承载数据构成。长度是包的总大小，头大小包括在数据大小内。当你的应用是 UDP 包决定缓冲大小，你应该记住 `socket.send` 长度仅仅是 `buffer`（有效负载），而且总的包大小必须在 MTU 范围内。数据报结构就如下图所示。



UDP 头 8 字节，接着是可选的数据负载，IPv4 上限 65507，IPv6 是 65527

服务端比客户端更简单。它用同样的方式创建了 `socket`^⑦，然后订阅了两个事件。第一个事件是 `message`，当数据报接收到的时候触发^⑧。数据通过使用 `process.stdout.write` 写到终端。这看上去要比使用 `console.log` 好，因为它不会自动增加新行。

当服务端准备接收连接时触发 `listening` 事件^⑨。为表明这一点会显示一条消息，因此你会知道尝试连接一个客户端是安全的。

虽然这是一个简单的例子，但是显而易见地说明 UDP 为什么不同于 TCP——你需要注意到你发送消息的大小，并且意识到消息是会丢失的。即使数据报有 `checksum`，丢失或损坏的包并不会告知应用层，这就意味着数据丢失是很有可能。通常的做法最好是使用 UDP 发送数据，相对于高速和吞吐量确保完整性是屈居第二的。

在下一个技巧中，你将会看到如何构建通过发送消息给客户端的例子，实际上是使用 UDP 建立双向通道。

技巧 49 UDP 客户端服务应用

UDP 经常被用来查询-响应协议，比如 DNS 和 DHCP。这个技术展示了如何将消息发送回客户端。

问题

你已经创建了一个 UDP 服务来响应和请求，但是你想要将消息发送回客户端。

解决方案

一旦你创建了一个服务，并且它已经接收到了消息，创建了一个数据报连接返回客户端是基于传到 `message` 事件的 `info` 参数。通过连接客户端端口和 IP 地址发送连续消息来选择性地创建一个唯一参考。

讨论

聊天服务器对 Node 新手来说是典型的网络程序示例，但是这里有个小坑——它使用 UDP 替代 TCP 或 HTTP。

TCP 连接与 UDP 是不同的，而且在 Node 网络 API 设计里是很明显的。TCP 连接被表示为一个双向事件流，因此发回一个消息到发送端很简单。一旦一个客户端已经连接，你能够使用 `client.write` 向它写消息。在另一方面，UDP 是非面向连接的——不需要有效的连接就可以接收消息。

这有一些级别相似的协议，使你能够从客户端响应数据。TCP 和 UDP 连接使用源和目标端口。给定一个合适的网络设置，打开一个基于这个信息的客户端连接。在 Node 中，`rinfo` 对象包含每个 `message` 事件，包含相关细节。图 7.5 中展示了使用这个方案如何在两个客户端中传递数据流。

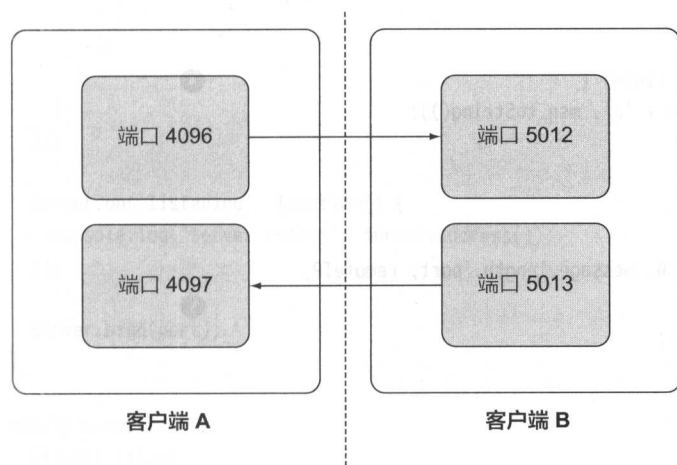


图 7.5 尽管 UDP 不是完全双向的，当两边在两个方向都提供一个端口号时，它可以创建双向连接

例子 7.5 介绍了一个客户端-服务端程序，允许客户端通过 UDP 连接到一个中心服务器，并互相通信。服务端在一个数组里保存了每一个客户端，因此它能够独立地映射每一个。通过保存客户端地址和端口，你甚至能够在同一个机器运行多个客户端——在同一个电脑多次运行这个程序是安全的。

例子 7.5 把消息发送回客户端

```
var assert = require('assert');
var dgram = require('dgram');
var fs = require('fs');
var defaultSize = 16;
var port = 41234;

function Client(remoteIP) {
  var socket = dgram.createSocket('udp4');
  var readline = require('readline');
  var rl = readline.createInterface(process.stdin, process.stdout);

  socket.send(new Buffer('<JOIN>'), 0, 6, port, remoteIP);

  rl.setPrompt('Message> ');
  rl.prompt();

  rl.on('line', function(line) {
    sendData(line);
  }).on('close', function() {
    process.exit(0);
  });

  socket.on('message', function(msg, rinfo) {
    console.log('\n<' + rinfo.address + '>', msg.toString());
    rl.prompt();
  });

  function sendData(message) {
    socket.send(new Buffer(message), 0, message.length, port, remoteIP,
      function(err, bytes) {
        console.log('Sent:', message);
        rl.prompt();
      }
    );
  }
}

function Server() {
  var clients = [];
  var server = dgram.createSocket('udp4');

  server.on('message', function(msg, rinfo) {
```

```
var clientId = rinfo.address + ':' + rinfo.port;

msg = msg.toString();

if (!clients[clientId]) {
  clients[clientId] = rinfo;
}

if (msg.match(/^</)) {
  console.log('Control message:', msg);
  return;
}

for (var client in clients) {
  if (client !== clientId) {
    client = clients[client];
    server.send(
      new Buffer(msg), 0,
      msg.length, client.port, client.address,
      function(err, bytes) {
        if (err) console.error(err);
        console.log('Bytes sent:', bytes);
      }
    );
  }
}

server.on('listening', function() {
  console.log('Server ready:', server.address());
});

server.bind(port);
}

module.exports = {
  Client: Client,
  Server: Server
};

if (!module.parent) {
  switch (process.argv[2]) {
    case 'client':
      new Client(process.argv[3]);
      break;
  }
}
```

```
case 'server':  
  new Server();  
  break;  
  
default:  
  console.log('Unknown option');  
}  
}
```

- ❶ 使用 `readline` 模块来处理用户输入。
- ❷ 无论何时，一个客户端加入时，发送特殊的加入消息。
- ❸ 当用户输入一个消息后按下回车键，便发送消息给服务器。
- ❹ 监听其他用户的消息。
- ❺ 获取用户的消息，并且创建一个新的 `buffer` 来作为 UDP 消息发送给服务器。
- ❻ 监听客户端的新消息。
- ❼ 结合客户端的端口号和地址来创建一个唯一的引用。
- ❽ 如果这个客户端之前没见过，那么记录它连接的细节信息。
- ❾ 如果消息是用尖括号包裹起来的，那么便是控制消息。
- ❿ 发送消息给其他每一个客户端。

这个例子基于技巧 48——你能够用同样的方式运行它。输入 `node udp-chat.js server` 来启动一个服务，然后 `node udp-chat.js client` 来连接一个客户端。你应该运行一个以上来工作；否则信息不会路由到任何地方。

`readline` 模块被用来友好地捕获用户输入❶。就像大多数你见过的核心模块，这是基于事件的。当一行文本进入时，它会触发 `line` 事件❸。

在消息能够被用户发送前，一个初始 `join` 消息会被发送❷。这只是为了让服务端知道它已经连接，服务端代码使用它来存储一个客户端独立的引用❽。

`Client` 构造函数内部封装了 `socket.send`，为 `sendData`❺。这就是为什么一行文本输入消息就会轻松地发送。同样，当客户端自己接收到一个消息，它将会打印到控制台，并通过创建服务器接收到一个新的提示❹。

通过信息❻所使用的组合端口和远程地址❼，创建一个唯一的客户。我们从 `rinfo` 对象中获得所有信息，而且在同一个机器运行多个客户端是安全的，因为端口是客户端的而不是服务端监听的（不能改变）。为了理解如何实现，重新调用 UDP 头，包含一个像 TCP 那样的源和目标端口。

最后，当消息不是控制消息⑨，每个客户端被迭代和发送消息⑩。已发送的邮件客户端将不会收到副本。因为我们已经存储引用，以在客户端列出各 rinfo 对象，信息可以发送回客户端。

客户端-服务端网络是 HTTP 的基础。即使 HTTP 使用 TCP 连接，从你目前能看到的协议类型来看稍有不同：它是无状态的。这意味着你需要用不同的模式来模拟它。下一节有更多详细内容讨论 HTTP 客户端和服务端。

7.4 HTTP 客户端和服务端

今天，无论我们生产或者消费 Web 服务，或者构建 Web 应用，大多使用 HTTP。HTTP 协议是无状态的，而且基于 TCP，并且 Node 的 HTTP 模块更像是基于 TCP 模块构建的。

当然，你能使用你自己的协议构建 TCP。然后 HTTP 基于 TCP。但是由于网络浏览器和工具在处理基于 Web 的服务工作方面的流行，HTTP 在处理远程系统通信方面自然适合。

下面你将会学习如何使用 Node 核心模块写一个基本的 HTTP 服务。

技巧 50 HTTP 服务器

在这个技术中，你将会学习如何使用 http 模块创建 HTTP 服务。尽管这比使用 Node 的 Web 框架要多很多工作，流行的 Web 框架内部通常使用相同的技术实现，它们暴露的对象都来源于 Node 的标准类。因此了解底层的模块和类对于广泛使用 HTTP 是有用的。

问题

你想要运行 HTTP 服务器以及对它们进行测试。

解决方案

使用 `http.createServer` 和 `http.createClient`。

讨论

`http.createServer` 方法是从 `net.Server` 创建一个新的 `http.Server` 对象的捷径。HTTP 服务被扩展成为支持多种 HTTP 协议的元素，解析头部信息，处理响应码，并且设置 sockets 上的各种事件。Node HTTP 处理响应码的重点是解析；一个 C++ 封装过的 C 解析库被封装使用。这个库能提取头部和值、Content-Length、请求方法响应状态码和更多东西。

例子 7.6 展示了一个小的“Hello World”Web 服务，使用了 http 模块。

例子 7.6 一个简单的 HTTP 服务器

```
var assert = require('assert');  
var http = require('http');  
  
var server = http.createServer(function(req, res) {  
    res.writeHead(200, { 'Content-Type': 'text/plain' });  
    res.write('Hello, world.\r\n');  
    res.end();  
});  
  
server.listen(8000, function() {  
    console.log('Listening on port 8000');  
});  
  
var req = http.request({  
    port: 8000  
}, function(res) {  
    console.log('HTTP headers:', res.headers);  
    res.on('data', function(data) {  
        console.log('Body:', data.toString());  
        assert.equal('Hello, world.\r\n', data.toString());  
        assert.equal(200, res.statusCode);  
        server.unref();  
    });  
});  
  
req.end();
```

- ❶ 加载 HTTP 模块。
- ❷ 创建新的 HTTP 服务器，传递一个回调，它会在一个新的请求到来时执行。
- ❸ 写入基于文本的响应头。
- ❹ 发送消息回客户端。
- ❺ 设置服务器监听 8000 端口。
- ❻ 使用 `http.request` 来创建请求。
- ❼ 给 `data` 事件添加监听器，来确保响应和所期望的是一样的。

`http` 模块包括 Node 客户端和 HTTP 服务端❶。`http.createServer` 创建了一个新的 `server` 对象并且将它返回。参数是一个回调，用来接受 `req` 和 `res` 对象——分别是请求和响应❷。如果你使用像 `Express` 和 `restify` 这样高级 Node web 框架，你可能对这些对象比较熟悉。

有趣的是关于传入到 `http.createServer` 的监听回调的行为与传给 `net.createServer` 的很像。的确，机制是相同的——我们创建 TCP sockets，但是在 HTTP 层上。HTTP 协议和 TCP socket 协议主要概念上的不同是状态的问题：HTTP 是无状态协议。这是完全可以接受的，而且事实上每次请求都要释放 TCP sockets。

在例子 7.6 里，监听器在每次请求时运行。在技巧 45 的 TCP 示例中，服务在客户端连接的过程中保持开启。因为 HTTP 连接仅仅是 TCP sockets，我们能够像例子 7.6 中的那样使用 `res` 和 `req`：`res.write` 将会写入 socket^④，并且头部能够使用 `res.writeHead`^③ 写回，这也是 socket 连接和 HTTP APIs 的明显分歧——底层的 socket 将会在响应写入的时候关闭。

在服务被创建后，我们能够设置它来用 `server.listen`^⑤ 监听一个端口。

现在我们能够创建服务器，再来看下创建 HTTP 请求。`http.request` 方法将会创建一个新连接^⑥，并且接受 `options` 参数对象和一个在连接建立时执行的回调。这就意味着我们仍然需要给响应附上一个 `data` 监听器传给回调获得任何发送数据。

`data` 回调确保来自服务端的响应拥有预期的格式：内容体和状态码^⑦被检查过。当最后一个客户端通过调用 `server.unref` 断开连接的时候服务被告知停止监听连接，这意味着脚本退出干净。这使得如果遇到这个问题很容易被发现。

HTTP 模块的小特性是 `http.STATUS_CODES` 对象。它允许通过查找整数状态码生成可读的消息：`http.STATUS_CODES[302]` 将会被认为临时移除。

现在你可以看到如何创建 HTTP 服务器，在下一个技巧中，我们将会看一下在 HTTP 客户端中的角色状态——尽管 HTTP 是一个无状态协议——通过实现 HTTP 重定向。

技巧 51 重定向

Node `http` 模块提供一个实用的 API 处理 HTTP 请求。但是它不能处理重定向，因为重定向在 Web 中很常见，它是个主流技术。你能够使用一个三方模块处理重定向，像流行的请求模块 Mikeal Rogers，²但是你会通过查看它如何使用核心模块实现来学习更多关于 Node 的内容。

在这个技巧中我们将会看一下如何使用 JavaScript 来维护跨多个请求的状态。这使得重定向被正确执行，而无须创建重定向循环或其他什么。

²<https://npmjs.org/package/request>

问题

你想要下载页面并且在需要的时候重定向。

解决方案

一旦协议的基础被理解，处理重定向是相当简单的。HTTP 标准定义了表示重定向发生时的状态码，而且它也指出客户端应该检测无限循环。为了满足这些需求，我们将使用一个简单的原型类，保留每个请求的状态、重定向和重定向检测循环。

讨论

在这个例子中将使用 Node 的核心 http 模块来创建一个 GET 请求，我们事先知道将会产生重定向。如果要判断一个给定的响应是一个重定向，需要检查是否返回的状态代码开始用 3。所有的状态码是 3xx，指示已发生某种类型的重定向。

根据该规范，这是一整套我们需要处理的状态代码：

- 300——多重选择
- 301——永久移动到新位置
- 302——找到重定向跳转
- 303——参见其他信息
- 304——没有改动
- 305——使用代理
- 307——临时重定向

究竟如何处理每个状态代码依赖于应用的实现。例如，它可能是非常重要的，一个搜索引擎以标识返回 301 响应，因为这意味着 URL 的搜索引擎的列表应该永久更新。对于本技巧，我们只需要进行重定向，这意味着一个语句就足以检查请求是否被重定向：if (response.statusCode >= 300 && response.statusCode < 400)。

测试重定向循环更加复杂。一个请求在隔离中不复存在——我们需要跟踪多个请求的状态。模拟这种最简单的方法是使用一个类，其中包括用于计算有多少重定向发生的实例变量。当计数器达到一个极限，将引发错误。图 7.6 显示了 HTTP 重定向怎么处理。

在写代码之前，想好需要哪些 API 很重要。既然已经决定将会用一个“类”管理状态，那么我们模块的用户将需要实例化这个类的实例。Node 的 http 模块是异步的，你的代码也应该这样。那就意味着获得一个返回结果，我们不得不传递一个回调方法。

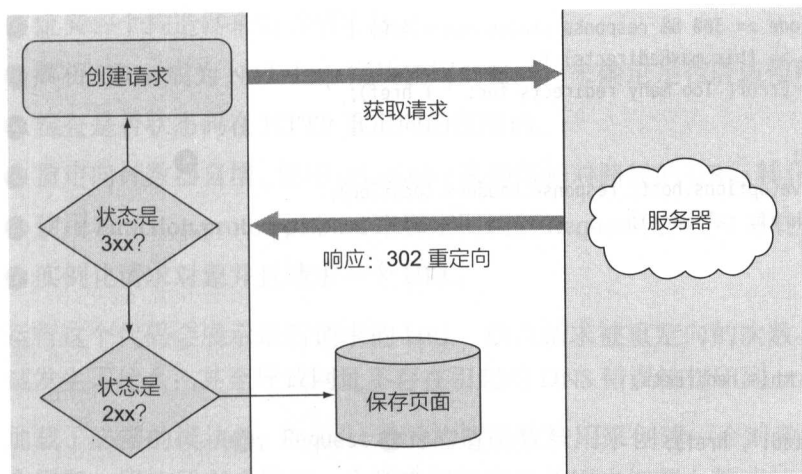


图 7.6 重定向是可以循环的，请求会一直创建直到出现 200 状态

对于此回调的签名应该使用相同的格式作为 Node 的核心模块，第一个参数是错误的变量。这种方式设计的 API 有处理错误直观的优点。创建一个 HTTP 请求能够获得很多错误，因此正确处理它们很重要。

下面的例子把这些集中起来便于成功地重定向。

例子 7.7 创建一个 HTTP 的 GET 请求来处理重定向

```

var http = require('http');
var https = require('https');
var url = require('url');
var request;

function Request() {
  this.maxRedirects = 10;
  this.redirects = 0;
}

Request.prototype.get = function(href, callback) {
  var uri = url.parse(href);
  var options = { host: uri.host, path: uri.path };
  var httpGet = uri.protocol === 'http:' ? http.get : https.get;

  console.log('GET:', href);

  function processResponse(response) {

```

```
if (response.statusCode >= 300 && response.statusCode < 400) {
  if (this.redirects >= this.maxRedirects) {
    this.error = new Error('Too many redirects for: ' + href);
  } else {
    this.redirects++;
    href = url.resolve(options.host, response.headers.location);
    return this.get(href, callback);
  }
}

response.url = href;
response.redirects = this.redirects;

console.log('Redirected:', href);

function end() {
  console.log('Connection ended');
  callback(this.error, response);
}

response.on('data', function(data) {
  console.log('Got data, length:', data.length);
});

response.on('end', end.bind(this));

httpGet(options, processResponse.bind(this))
  .on('error', function(err) {
    callback(err);
  });

};

request = new Request();
request.get('http://google.com/', function(err, res) {
  if (err) {
    console.error(err);
  } else {
    console.log('Fetched URL:', res.url,
      'with', res.redirects, 'redirects');
    process.exit();
  }
});
```

❶ url 模块有很多用于解析 URLs 的方法。

- ❷ 定义一个构造器来管理请求状态。
- ❸ 解析 URLs 成为 Node http 模块使用的格式，来确定是否应该使用 HTTPS。
- ❹ 检查是否状态码在 HTTP 重定向的范围内。
- ❺ 重定向计数器自增，使用 `url.resolve` 来确保相对路径的 URLs 转化为绝对路径的 URLs。
- ❻ 使用 `Function.prototype.bind` 来绑定回调到 `Request` 实例上，来让 `this` 指向正确的对象。
- ❼ 实例化请求对象并且请求一个 URL。

运行这个代码会展示最后请求的 URL，以及请求被重定向的次数。使用一些 URL 来尝试发生了什么：甚至导致网址不存在引起的 DNS 错误被打印到 `stderr`。

加载了必需的模块❶，`Request` ❷的构造函数被用来创建一个对象来构成请求对象的生命周期。用这种方式使用一个类来保持实现从用户细节上整洁的包装。与此同时，`request.prototype.get` 方法做了最多的工作。它建立了一个 HTTP 请求的标准，如果需要的话也可以是 HTTPS，并且遇到重定向时调用自身递归。注意 URL 已经被解析成一个我们用来创建兼容 http 模块的 `options` 对象❸。

请求协议（HTTP or HTTPS）被检测来确保我们使用 Node http 或 https 模块正确的方法。一些服务器被配置为了将 HTTP 总是重定向到 HTTPS。不检查的协议，这个方法会重复获取原始 HTTP URL，直到 `maxRedirects` 被触发，这是一个微不足道的错误，这是很容易一次性避免的。

`statusCode` 应该被检查❹。重定向次数会随着 `maxRedirects` 没有被触发而增加起来❺。这个过程被重复执行直到没有 300 段的状态，或者太多重定向被触发。

当最后的请求被完成（或者如果第一个没有重定向），用户提供的 `callback` 将会被运行。标准 Node API 错误 `error`、`result` 被用在这与 Node 核心模块一致。达到 `maxRedirects` 数量时触发一个错误，或者当创建一个 HTTP 请求时则通过监听一个 `error` 事件。

用户提供的回调在最后一次请求完成时运行，允许回调函数里访问请求资源。这是通过运行回调结束事件的最后一个请求已被触发后，并且由事件处理程序绑定到当前 `Request` 实例❻处理。绑定事件处理程序意味着它将能够访问到任何用户会需要的有用的实例变量——包括 `this.error` 中的错误。

最后，创建一个 `Request` 实例❼来试探这个类。如果你喜欢，你能够使用其他 URL。

本技术插图一个重要点：状态是重要的，即使 HTTP 是个无状态协议。一些错误配置的 web 应用程序和服务器可以创建重定向循环，这将导致客户端一直获取 URL，直到它被强制停止。

尽管例子 7.7 显示了一些 Node HTTP 和 URL 处理特性,但它不是一个完整的解决方案。想要一个更进一步的 HTTP API,看一下 Mikeal Rogers (<https://github.com/mikeal/request>),被广泛使用的简单 Node HTTP API。

在下一个技巧中,我们将会剖析一个简单的 HTTP 代理。对于客户端和服务端技术的这项扩展将在这里讨论,并且可以扩展到创建许多有用的应用。

技巧 52 HTTP 代理

HTTP 代理可能会比你想象中常用——ISP 使用透明代理,使网络更加高效,企业系统管理员使用缓存代理服务器来减少带宽,web 应用程序的 DevOps 利用它们来提高应用程序的性能。本技术只触及代理的表面——它获取 HTTP 请求和响应,然后将其传送到它们该去的地方。

问题

你想要获取并且转发 HTTP 请求。

解决方案

使用 Node 内置的 HTTP 模块作为一个简单的 HTTP 代理。

讨论

一个代理服务器提供一定程度的重定向,会促进多种有用的应用:缓存、日志和安全相关的软件。本技巧探索了如何使用核心的 `http` 模块创建 HTTP 代理。从根本上所有那些需要的是一个 HTTP 服务器来获取请求,然后一个 HTTP 客户端来复制它们。

`http.createServer` 和 `http.request` 方法能够获取并且转发请求。我们也需要解析原请求,因此能够安全拷贝它——`url` 核心模块有一个 `URL`——解析方法能够帮助我们做这些。

例子 7.8 展示了用 Node 创建一个工作的代理是多么简单。

例子 7.8 使用 `http` 模块来创建代理

```
var http = require('http');
var url = require('url');
```

```
http.createServer(function(req, res) {
  console.log('start request:', req.url);
  var options = url.parse(req.url);
  options.headers = req.headers;
  var proxyRequest = http.request(options, function(proxyResponse) {
    proxyResponse.on('data', function(chunk) {
```

1

2

3

```
    console.log('proxyResponse length:', chunk.length);
    res.write(chunk, 'binary');
  });

  proxyResponse.on('end', function() {
    console.log('proxied request ended');
    res.end();
  });

  res.writeHead(proxyResponse.statusCode, proxyResponse.headers);
});

req.on('data', function(chunk) {
  console.log('in request length:', chunk.length);
  proxyRequest.write(chunk, 'binary');
});

req.on('end', function() {
  console.log('original request ended');
  proxyRequest.end();
});
}).listen(8080);
```

- ❶ 创建标准的 HTTP 服务器实例。
- ❷ 创建请求来复制原始的请求。
- ❸ 监听数据，然后返回给浏览器。
- ❹ 追踪代理请求什么时候完成。
- ❺ 发送头部信息给服务器。
- ❻ 捕获从浏览器发送到服务器的数据。
- ❼ 追踪原始的请求什么时候结束。
- ❽ 监听来自本地浏览器的连接。

为了只用这个例子，你的计算机将需要一个小配置。找到你的系统网络选项，然后找到 HTTP 代理。从这里你将能够进入 `localhost:8080` 作为代理。另外，如果有需要，要在浏览器设置里添加代理。一些浏览器不支持这个功能；Google Chrome 将会打开系统代理对话框。

图 7.7 展示了如何在 Mac 上设置代理。确保你在主网络对话框点击了 OK 然后应用保存设置。并且记住一旦完成之后禁用代理！

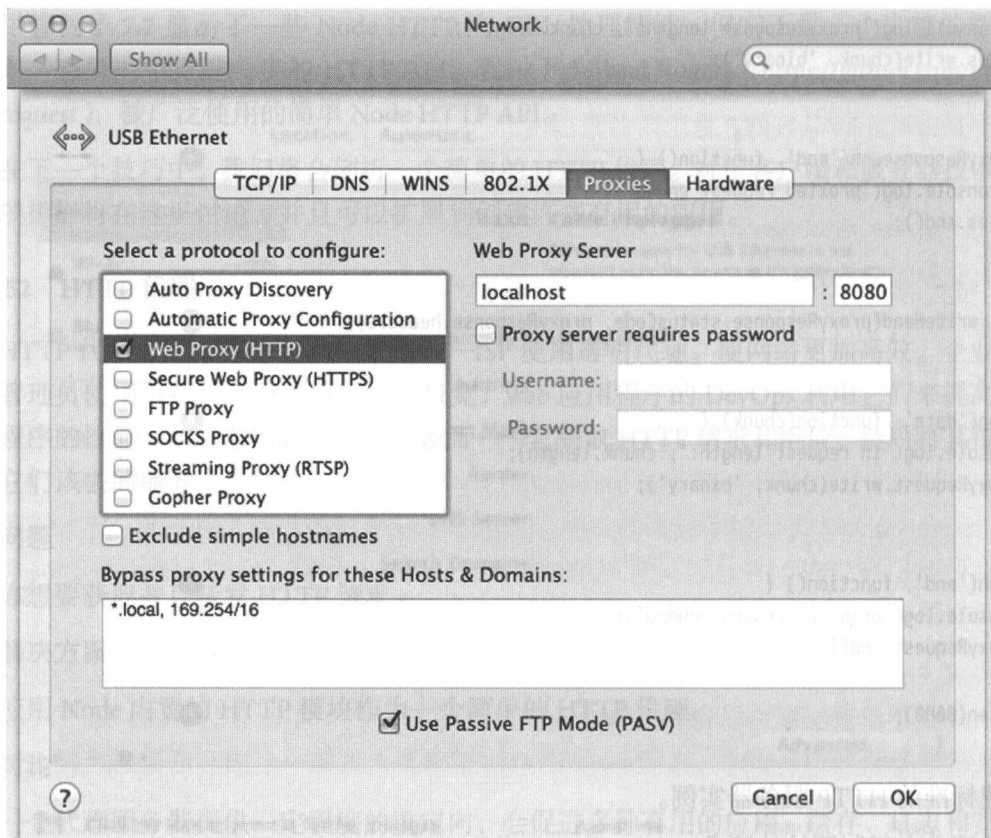


图 7.7 设置 web 代理服务器为 localhost:8080 来使用我们创建的 Node 代理

一旦你的系统被设置使用代理,在 shell 里使用 `node listings/network/proxy.js` 启动 Node 进程。现在访问网页,你应该看到登录到控制台连续请求和响应。

这个例子的工作原来是首先使用 `http` 模块创建一个服务器^①。当一个浏览器创建请求回调将会被执行。我们使用了 `url.parse` (`url` 是另一个核心模块)来分离出 URL 的变量部分,因此它们能够被作为参数传递给 `http.request`。被解析过的 URL 对象是兼容 `http.request` 参数预期的,因此很方便^②。

在请求的回调内,我们能够订阅重复返回浏览器的事件。`data` 事件是有用的,因为它允许我们从服务端捕获请求并且通过 `res.write` 传回客户端^③。我们也通过关闭浏览器连接来响应到服务端连接^④。基于服务端的状态码也被写回客户端^⑤。

任何通过客户端发送的数据也通过订阅浏览器 `data` 事件代理到远程服务端^⑥。同样,浏览器原请求被看作 `end` 事件,因此它能被反射回代理请求^⑦。

最后，用作代理的 HTTP 服务被设置为监听 8080 端口^⑧。

这个例子在浏览器和要访问的服务器之间创建了一个特殊的服务器。它能够被扩展来做很多有趣的事情。例如，你能够缓存图片文件和基于远程客户端压缩它们，发送压缩过的图片给手机浏览器。你甚至能通过规则去掉某些部分；一些广告拦截和家长过滤通过这种形式来做。

我们已经使用 DNS 但至今没有认真思考过。DNS 使用 TCP 和 UDP 作为它的请求/响应协议。幸运的是，Node 为我们隐藏了这些复杂的部分而机智地提供了异步 DNS 模块。下一节展示了如何使用 dns 模块创建 DNS 请求。

7.5 创建 DNS 请求

Node 的 DNS 模块是在 net 模块之外，在 dns 里。当 http 或 net 模块被用来连接远程服务，Node 将会使用内部的 dns.lookup 查找 IP 地址。

技巧 53 创建 DNS 请求

Node 有多种方式创建 DNS 请求。在本技巧中你将会学习到如何和为什么将会使用每个方式来解析域名到一个 IP 地址。

当查询一个 DNS 记录，结果可能包含不同的记录类型。DNS 是一个分布式数据库，因此它不是被用来单纯地解析 IP 地址——一些记录像 TXT 被用来构建 DNS 自身后的功能。

表 7.2 包含了每种类型及相关的 dns 模块方法。

表 7.2 DNS 记录类型

Type	Method	Description
A	dns.resolve	一个 A 记录存储 IP 地址。它可以有一个关联的生存时间（TTL）来表示这个记录多久需要进行更新
TXT	dns.resolveTxt	文本值可以用于在 DNS 上构建的其他服务的额外特性
SRV	dns.resolveSrv	服务记录定义服务的定位数据，通常它会包括主机名称和端口号
NS	dns.resolveNs	用于指定域名服务器
CNAME	dns.resolveCname	相关的域名记录，设置为域名而不是 IP 地址

问题

你想要快速查询一个或多个域名。

解决方案

`dns.lookup` 方法能够被用来查找 IPv4 或 IPv6 地址。当找到多个地址, 它能够被快速替换为 `dns.resolve`。

讨论

依据 Node 文档, `dns.lookup` 是通过一个线程池, 而 `dns.resolve` 关心使用更快的库。`dns.lookup` API 更加友好——它使用 `getaddrinfo`, 这与我们系统上的其他程序更加一致。确实, `Socket.prototype.connect` 方法, 和一些继承自 `net` 模块的 Node 的核心模块, 都一致地使用 `dns.lookup` :

```
var dns = require('dns');  
  
dns.lookup('www.manning.com', function(err, address) {  
  if (err) {  
    console.error('Error:', err);  
  }  
  console.log('Addresses:', address);  
});
```

❶ 加载 `dns` 模块。

❷ 通过给定的域名确定 IP 地址。

这个例子加载了 `dns` 模块❶, 然后使用 `dns.lookup` 查找 IP 地址❷。API 是异步的, 因此我们不得不传递一个回调来接收 IP 地址和查找地址产生的一些错误。注意, 域名必须被提供, 而不是一个 URL——这里不包含 `http://`。

如果一切正确运行, 那么你将会看到 `68.180.151.75` 作为 IP 地址打印。相反, 如果离线运行前面的例子, 那么将会打印一个有趣的错误:

```
Error: {  
  [Error: getaddrinfo ENOTFOUND]  
  code: 'ENOTFOUND',  
  errno: 'ENOTFOUND',  
  syscall: 'getaddrinfo'  
}
```

❶ 错误码。

❷ 错误产生的系统调用。

错误对象包括标准错误码❶, 是由于系统调用产生的错误❷。你可以在你的程序中使用错误码来检测和适当地处理错误。系统调用, 与此同时, 对于程序员是有用的: 它展示

了错误是 Node 代码之外操作系统的服务产生的。

现在比较使用 `dns.resolve` 和这个版本：

```
var dns = require('dns');

dns.resolve('www.manning.com', function(err, addresses) {
  if (err) {
    console.error(err);
  }

  console.log('Addresses:', addresses);
});
```

❶

❶ 异步地解析域名。

这个 API 看起来像之前的例子，除了 `dns.resolve` ❶。你仍然将会看见一个错误对象包含了 `ECONNREFUSED`，如果 DNS 服务不能使用，但是这结果会有所不同：我们接收到一个地址数组而不是一个单独的结果。例子中你将会看见 ['68.180.151.75']，但是有些服务可能会返回不止一个地址。

Node 的 `dns` 模块是灵活的、友好的、快速的。它能够从单个请求到大量请求很好地增长。

Node 网络套件最后的部分或许是最难学习的，但也是最重要的：加密。下一节介绍 `SSL/TLS`，`tls` 和 `https` 模块。

7.6 加密

Node 的加密模块，`tls`，使用了 `OpenSSL` 安全传输层套接字（`TLS/SSL`）。这是一个公开密钥系统，其中每一个客户端和服务端都有一个私钥。服务器使公钥可用，以便客户端可以加密，只有该服务器可以再次解密方式后续通信。

`tls` 模块被用作 `https` 模块的基础模块——它允许 HTTP 服务端和客户端通过 `TLS/SSL` 通信。不幸的是，`TLS/SSL` 是一个充满坑的世界。Node 可能支持基于不同版本 `OpenSSL` 的 `cyphers`。你可以指定你想与 `tls.createServer` 创建服务器时使用什么 `cyphers`，但建议使用默认值，除非你有这方面的专门知识。

在下一个技巧中，你将会学习到如何使用 `SSL` 和一个自签名证书启动一个 `TCP` 服务。然后，在结尾我们有一个板块展示了如何使用 Node 与服务端加密通信。

技巧 54 一个加密的 TCP 服务器

TLS 能够被用来使用 `net.createServer` 加密服务端。这个技术展示了如何创建一个必要的证书, 然后开启一个客户端和一个服务端。

问题

你想要通过 TCP 连接来加密发送和接收通信。

解决方案

使用 `tls` 模块开启一个客户端和一个服务端。使用 `OpenSSL` 创立一个所需要的证书文件。

讨论

使用加密工作的主要事情是, 无论是不是 Web 服务器、邮件服务器, 或者任何基于 TCP 协议的, 是如何正确地设置密钥和证书文件。公钥加密依赖于公钥-私钥对——客户端和服务端需要一对。但是一个附加文件也是需要的: 证书验证公钥 (CA)。

这个技巧的目标是在 TLS 握手后创建一个 TLS 客户端和服务端。报告这种状态时, 双方已经验证对方身份。当使用 Web 服务端证书工作, 你的 CA 将会被公职的组织颁发。但是对于测试目的, 你能够为自己的 CA 签署证书。这也易于自己的系统之间的安全通信不需要公开验证的证书。

这就意味着在运行任意 Node 示例前, 将会需要证书。这就需要 `OpenSSL` 命令行工具。如果没有他们, 你应该使用操作系统的包管理或者通过访问 www.openssl.org 安装。

`openssl` 工具接受命令作为第一个参数, 然后选择随后的参数。例如, `openssl req` 用于 X.509 证书签名请求 (CSR) 的管理。为了让你通过一个控制中心签署的证书, 则需要发出以下命令:

- `genrsa`——生成一个 RSA 证书; 这是我们的私钥。
- `req`——创建一个 CSR。
- `x509`——使用 CSR 产生一个公钥签署私钥。

当过程像这样分解, 就很容易理解: 证书要求一个权威且必须签名, 我们需要一个公钥和一个私钥。过程很类似于创建签署商业认证机构, 如果你想购买证书, 公共 Web 服务器使用公钥和私钥时你会怎么做。

创建一个公钥和私钥完整的命令如下:

```
openssl genrsa -out server.pem 1024
openssl req -new -key server.pem -out server-csr.pem
```

1

2

```
openssl x509 -req -in server-csr.pem -signkey server.pem \
-out server-cert.pem
```

3

```
openssl genrsa -out client.pem 1024
```

4

```
openssl req -new -key client.pem -out client-csr.pem
```

5

```
openssl x509 -req -in client-csr.pem -signkey client.pem \
-out client-cert.pem
```

6

❶ 使用 1024 比特创建服务器的私钥。

❷ 创建 CSR，这里要输入主机名。

❸ 签发服务器的私钥。

❹ 创建客户端的私钥。

❺ 为客户端创建 CSR，记住这里也要输入主机名。

❻ 签发客户端的私钥，然后输出一个公钥。

创建一个私钥❶，你将要创建一个 CSR。当提示“Common Name”❷，输入你的计算机的主机名，可以通过在 UNIX 系统终端输入 `hostname` 找到。这一点很重要，因为当你的代码发送或接收证书，它会对证传递给 `tls.connect` 方法 `servername` 属性的名称值。

例子 7.9 会读取服务器的密钥和使用 `tls.createServer` 启动服务器运行。

例子 7.9 一个使用 TLS 来加密的 TCP 服务器

```
var fs = require('fs');
var tls = require('tls');
```

```
var options = {
  key: fs.readFileSync('server.pem'),
  cert: fs.readFileSync('server-cert.pem'),
  ca: [ fs.readFileSync('client-cert.pem') ],
  requestCert: true
};
```

1

2

3

4

```
var server = tls.createServer(options, function(cleartextStream) {
  var authorized = cleartextStream.authorized ?
    'authorized' : 'unauthorized';
  console.log('Connected:', authorized);
  cleartextStream.write('Welcome!\n');
  cleartextStream.setEncoding('utf8');
  cleartextStream.pipe(cleartextStream);
});
```

5

```
server.listen(8000, function() {
  console.log('Server listening');
});
```

- ❶ 私钥。
- ❷ 公钥。
- ❸ 客户端验证证书。
- ❹ 确保客户端证书都要被检查。
- ❺ 当浏览器请求一个页面时, 展示服务器是否能够验证证书。

例子 7.9 中的网络代码与 `net.createServer` 方法是相似的——那是因为 `tls` 模块继承自它。代码的其余部分是有关管理证书的, 不幸的是这个过程是留给我们处理, 而且往往是程序员会错的地方, 这可能会危及安全的原因。首先, 加载了私人❶和公共❷组合键, 将它们传递给 `tls.createServer`。我们还加载客户端的公钥证书颁发机构❸——在使用获得的证书时, 这个阶段通常不要求。

当客户端连接, 我们希望发送一些数据, 但对于这个例子的目的, 我们真的只是想看看客户端是否被授权❺。客户端授权, 已被迫通过设置选项 `requestCert` ❹。

此服务器可以与 Node 运行 `node tls.js`, 但有一些缺失: 一个客户端! 接下来的例子包含了能够连接到该服务器的客户端。

例子 7.10 一个使用 TLS 的 TCP 客户端

```
var fs = require('fs');
var os = require('os');
var tls = require('tls');

var options = {
  key: fs.readFileSync('client.pem'),
  cert: fs.readFileSync('client-cert.pem'),
  ca: [ fs.readFileSync('server-cert.pem') ],
  servername: os.hostname()
};

var cleartextStream = tls.connect(8000, options, function() {
  var authorized = cleartextStream.authorized ?
    'authorized' : 'unauthorized';
  console.log('Connected:', authorized);
  process.stdin.pipe(cleartextStream);
});
```

❶

❷

❸

❹

❺

```
});

cleartextStream.setEncoding('utf8');

cleartextStream.on('data', function(data) {
  console.log(data);
});
```

- ❶ 加载私钥。
- ❷ 加载公钥。
- ❸ 服务端验证证书。
- ❹ 把主机名作为服务器名称。
- ❺ 从服务器读取数据后打印出来。

客户端类似于服务器：私钥❶和公钥❷加载，并且此时的服务器被视为 CA❸。服务器的名称被设置给 CSR 中通过使用 `os.hostname` ❹ 的值相同的通用名称——如果把它设置为别的东西，可以手动键入。在此之后，客户端连接，显示器是否能够授权证书，以及读取由服务器发送的数据并使管道将它置入到标准输出❺。

测试 SSL/TLS

当测试安全证书，很难知道问题是否出在你的代码或其他地方。围绕这一方法是使用 `openssl` 命令行工具来模拟一个客户端或服务器。下面的命令将启动用来连接与给定的证书文件服务器的客户端：

```
openssl s_client -connect 127.0.0.1:8000 \
➡ -CAfile ./server-cert.pem
```

`openssl` 工具将显示很多有关连接的额外信息。用在这个技巧中写的例子来弄清楚，我们会生成的证书有其通用名错误的值。

当你调用 `tls.createServer` 时 `tls`。

`Server` 的一个实例被实例化。这种构造方法调用 `net.Server` ——还有每个 `net` 模块之间有着明显的继承关系。这意味着通过 `net.Server` 触发的事件与 `TLS` 服务器的相同。

在接下来的技巧中，你将看到如何使用 `HTTPS`，怎么同时关联到 `tls` 和 `net` 模块。

技巧 55 加密的 Web 服务器和客户端

即使它可能承载像后面的 Apache 和 Nginx 的其他 Web 服务器 Node 的应用, 有的时候你会想运行自己的 HTTPS 服务器。本技巧引入了 https 模块, 并展示它是如何关联 tls 模块的。

问题

你想要运行一个支持 SSL/TLS 的服务器。

解决方案

使用 https 模块和 https.createServer。

讨论

在这个技术中运行例子, 你将会需要如下步骤创建合适的自签名证书, 就像技巧 54 中那样。一旦你创建了一些公钥和私钥, 你将能够运行这些例子。

例子 7.11 展示了一个 HTTPS 服务器。

例子 7.11 一个使用 TLS 加密的基础 HTTP 服务器

```
var fs = require('fs');
var https = require('https');

var options = {
  key: fs.readFileSync('server.pem'),
  cert: fs.readFileSync('server-cert.pem'),
  ca: [ fs.readFileSync('client-cert.pem') ],
  requestCert: true
};

var server = https.createServer(options, function(req, res) {
  var authorized = req.socket.authorized
    ? 'authorized' : 'unauthorized';
  res.writeHead(200);
  res.write('Welcome! You are ' + authorized + '\n');
  res.end();
});

server.listen(8000, function() {
  console.log('Server listening');
});
```

❶ 私钥。

- ② 公钥。
- ③ 确保客户端证书都要被检查。
- ④ 当浏览器请求一个页面时，展示服务器是否能够验证证书。

例子 7.11 中的服务器基本与技巧 54 一样加载并传递到 `https.createServer`。同样，私钥①和公钥②组合。

当浏览器请求一个页面，我们检查 `req.socket.authorized` 属性，查看是否要求被授权。这个状态被返回到浏览器。如果你想通过浏览器尝试一下，确保你键入 `https://` 到地址栏；否则将无法正常工作。你会看到一条警告消息，是因为浏览器将无法验证服务器的证书；你知道，因为你创建的服务器发生了什么事情。服务器将响应说，你是未经授权的，是因为它不能够授权。

为了使客户端能够连接到该服务器，参见如下代码。

例子 7.12 一个 HTTPS 客户端的例子

```
var fs = require('fs');
var https = require('https');
var os = require('os');

var options = {
  key: fs.readFileSync('client.pem'),
  cert: fs.readFileSync('client-cert.pem'),
  ca: [ fs.readFileSync('server-cert.pem') ],
  hostname: os.hostname(),
  port: 8000,
  path: '/',
  method: 'GET'
};

var req = https.request(options, function(res) {
  res.on('data', function(d) {
    process.stdout.write(d);
  });
});
req.end();

req.on('error', function(e) {
  console.error(e);
});
```

- ❶ 加载私钥。
- ❷ 加载公钥。
- ❸ 加载服务器的证书。
- ❹ 设置机器的名称来作为主机名。
- ❺ 使用 `https.request` 来创建 HTTPS 请求。

本示例设置私钥❶和公钥❷组合的客户端，这就是做透明的安全请求时你的浏览器做了什么。它还设置了服务器证书颁发机构❸，它通常不会被要求。用于 HTTP 请求的主机名是本机的主机❹。

一旦所有这种设置操作后，HTTPS 请求被允许。这是通过使用与 `https.request` ❺ 的 API 相同的 `http` 模块。在这个例子中，服务器将确保在 SSL/TLS 的授权程序是有效的，因此，服务器将返回文本，以指示在连接被充分授权。

在实际 HTTPS 代码中，你可能不会做自己的 CA。使用 HTTPS，只不过对于测试或通过互联网 API 请求的通信内部系统有用。当进行 HTTPS 对公共 web 服务器的请求，Node 将能够给你验证服务器的证书，这样你就不会需要设置密钥、证书和 CA。

`https` 的模块有一些其他功能——有一个 `https.get` 简便方法更容易让 GET 请求。相反，它封装了我们在 Node 加密模块中要用到的技术。

加密对

在将加密知识从草坪中移除之前，还需要细细咀嚼：`SecurePair`。这是一个 `tls` 模块中的类，能够被用来创建安全对流：一种读写加密数据，另一个读写明文。这可能允许你将任意数据通过流输出。

有一个简便方法是：`tls.createSecurePair`。当 `SecurePair` 建立安全连接时，它会发出安全事件，但你仍然需要检查 `cleartext.authorized` 以确保证书是适当的授权。

7.7 总结

本章很长，那是因为 Node 中网络部分是很重要的。Node 是建立在对网络编程优良基础上的；缓冲区、流和异步 I/O 都有助于一个非常适合写入下一代面向网络的程序的环境。

在本章，你应该能够体会 Node 如何适应网络软件的更广阔的世界。无论你正在开发 UNIX 守护进程，基于 Windows 的游戏服务器，或下一个庞大的 web 应用程序，你现在应该知道从哪里开始。

不用说，网络和加密密切相关。随着 Node 的 `tls` 和 `https` 模块，你应该能够编写网络客户端和服务端，可以与其他系统通信，而不用担心信息被窃取。

下一章将是最后的节点的核心模块，`child_process`，并着重于技术与其他命令行程序接口。

8 子进程：利用 Node 整合外部应用程序

本章概要

- 执行外部应用
- 分离一个子进程
- Node 进程间通信
- 可执行的 Node 程序
- 创建工作池
- 同步子进程

任何一个平台都不是孤立的。尽管通过 JavaScript 来编写一个平台的所有功能听上去是一件很有意思的事情，然而这么做必然使我们错过一些其他平台上现成的应用。举个例子来说，GraphicsMagick (<http://www.graphicsmagick.org>) 就是一个功能全面的图片操作工具，它可以很方便地用于调整已上传的大批量大型照片的大小。另外一个例子，就是 wkhtmltopdf (<http://wkhtmltopdf.org>)，它是一个基于没有界面的浏览器内核基础之上的一种 PDF 文件生成器，在处理将 HTML 文档转换成一个可下载的 PDF 文件方面表现得极佳。然而在 Node 的世界里，child_process 模块是允许我们在我们自己的 Node 程序

中执行诸如上述所说的一些外部程序的（当然这样的外部程序也包括一些利用 Node 开发的应用），这样一来，我们就不必要重新造轮子。

child_process 模块提供了四种不同的方法来执行外部应用程序。所有的这些方法也都是异步的。我们可以按照实际需要使用相应的方法，正如图 8.1 所示。

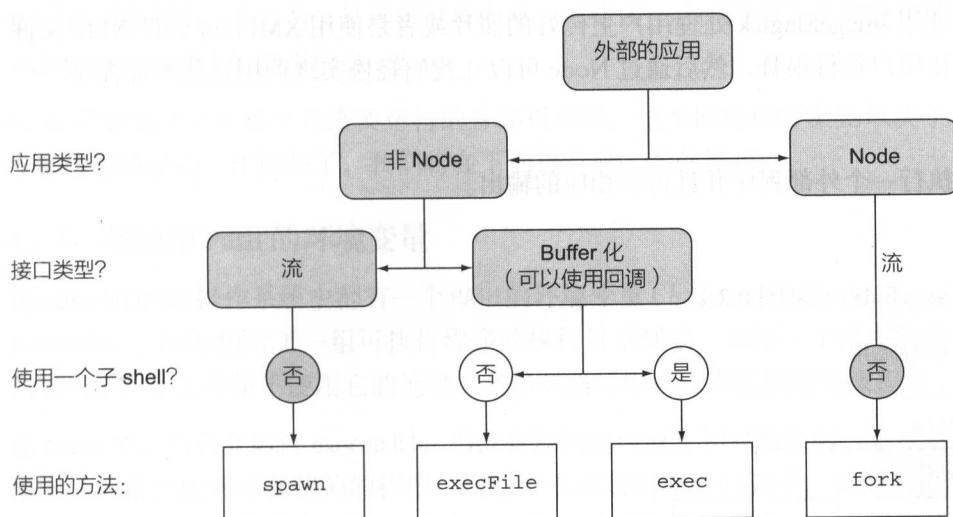


图 8.1 选择正确的方法

- **execFile**——执行外部程序，并且需要提供一组参数，以及一个在进程退出后的缓冲输出的回调。
- **spawn**——执行外部程序，并且需要提供一组参数，以及一个在进程退出后的输入输出和事件的数据流接口。
- **exec**——在一个命令行窗口中执行一个或者多个命令，以及一个在进程退出后缓冲输出的回调。
- **fork**——在一个独立的进程中执行一个 Node 模块，并且需要提供一组参数，以及一个类似 spawn 方法里的数据流和事件式的接口，同时设置好父进程和子进程之间的进程间通信。

在这一章中，我们会介绍如何充分地掌握利用这几种方法，并且为每一种你可能会使用的方法都提供相应的实例进行讲解，最后也会探讨一些其他关于子进程操作的技术，比如进程的分离、进程间通信、文件描述器和工作池。

8.1 执行外部应用程序

在章节的开始，我们一起来看看所有这些异步调用外部应用程序的方法。

技巧 56 执行外部应用程序

可以想象使用 ImageMagick 处理用户上传好的照片或者是使用 XMLLint 验证 XML 文件的格式都让用户觉得很赞，然后通过 Node 可以让我们轻松实现调用这些外部程序。

问题

你想通过执行一个外部程序并且得到相应的输出。

解决方案

通过使用 `execFile`（见图 8.2）。

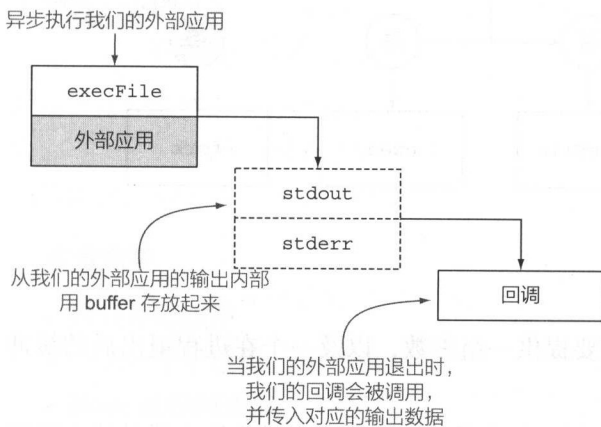


图 8.2 `execFile` 方法缓存结果并提供一个回调

讨论

如果你想通过运行一个外部的应用程序，并且得到相应的输出结果，那么使用 `execFile` 方法则是最直接了当的方法了。它会把输出结果为你缓存好，并且通过一个回调函数返回最后的结果或者异常信息。这么说吧，比如你想运行一个 `echo` 的命令程序，这个程序的输入参数是 `hello world`，使用 `execFile`，代码调用如下：

```
var cp = require('child_process');

cp.execFile('echo', ['hello', 'world'],
  function (err, stdout, stderr) {
```

```
if (err) console.error(err);
console.log('stdout', stdout);
console.log('stderr', stderr);
});
```

- ❶ 第一个参数是程序命令名称，第二个数组参数是命令输入。
- ❷ 回调函数里第一个参数，是运行过程中发生的错误，后面的两个参数包含了输出结果 `stdout` 和可能输出的错误信息 `stderr`。

Node 是如何知道从哪里找需要运行的外部程序呢，这个问题的答案则取决于不同的操作系统里路径的工作机制了，我们将在下面进行进一步的阐述。

8.1.1 路径和 Path 的环境变量

Windows/UNIX 操作系统中都有一个 PATH 的环境变量 ([http://en.wikipedia.org/wiki/PATH_\(variable\)](http://en.wikipedia.org/wiki/PATH_(variable)))。PATH 包含了一组可执行程序的执行目录列表。如果一个程序的路径在这个列表当中，那么即使不使用它的绝对或者相对路径也可以直接加载到该程序。

在 Node 中，当后台运行 `execvp` 时，当没有提供绝对或者相对路径时，它就会基于 PATH 里定义的路径搜索所有相关的程序。回过头来看我们前面的例子，类似 `echo` 这样的通用程序的路径一般都已经在 PATH 里定义好了，所以直接用程序名，也可以找到程序的执行目录。

如果执行的程序目录没有在 PATH 里预定义，那么执行时候就需要显式提供程序的具体执行路径，如：

```
cp.execFile('./app-in-this-directory' ...
cp.execFile('/absolute/path/to/app' ...
cp.execFile('../relative/path/to/app' ...
```

如果想要快速检查 PATH 路径包含哪一些目录，你可以在 Node 的交互式命令解析器 (REPL) 里输入一行简单的代码，如：

```
$ node
> console.log(process.env.PATH.split(':').join('\n'))
/usr/local/bin
/usr/bin/bin
...
```

如果既不想在 PATH 环境变量里包含你要执行的外部程序，又想通过前面一样方便的方式来调用外部程序，还有一种方法就是通过设置 `process.env.PATH` 来实现，但是你必须保证这个设置是在你执行 `execFile` 之前，如：

```
process.env.PATH += ':/a/new/path/to/executables';
```

这样一来，即使在 `execFile` 中不使用具体的路径，它也可以访问到所有在这个新的目录下的程序。

8.1.2 执行外部程序时候出现的异常

如果你需要调用的外部程序不存在，那么此时将会遇到一个 `ENOENT` 的错误，通常这样的错误都是因为我们把外部应用程序的名称或者是路径输入错了导致的，这样一来，Node 必然找不到这个需要被调用的外部应用，如图 8.3 所示。

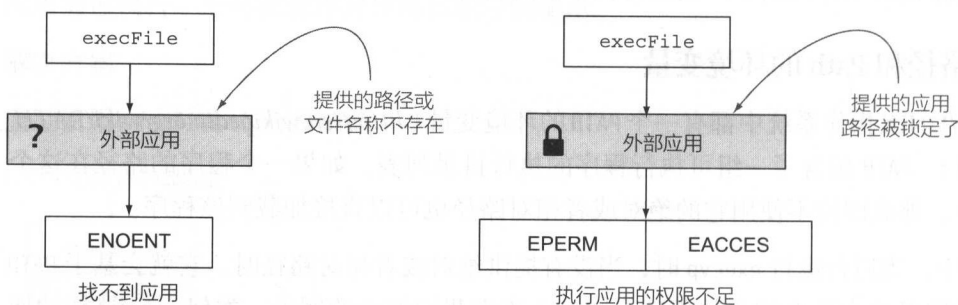


图 8.3 常见的子进程异常错误

另外一种情况，如果外部应用程序确实存在，但是 Node 却无法访问它（通常情况下都是因为没有足够的权限），此时你将会遇到一个 `EACCES` 或者是 `EPERM` 的错误。为了解决诸如这样的问题，你要么通过使用一个具有足够权限的用户来运行当前需要调用外部程序的 Node 的程序，要么对需要被调用的外部程序权限设置进行修改，使它可以被访问得到。

还有，当被调用执行的外部程序退出返回的状态码是非零状态时（<http://mng.bz/MLXP>），它表示该程序不能够在当前的平台（Windows 或者是 UNIX）下执行对应的任务。在这种情况下，Node 就会把该返回的状态码作为异常对象和其他一切可能返回的数据传入到 `stdout` 和 `stderr`，如下面的代码：

```
var cp = require('child_process');
cp.execFile('ls', ['non-existent-directory-to-list'],
  function (err, stdout, stderr) {
    console.log(err.code);
    console.log(stderr);
  });
```

1

2

❶ 设想, 此时 `code` 的值是 1, 表示命令执行失败。

❷ 具体的错误信息存储在 `stderr` 中。

综上所述, 当我们只是需要执行一个外部程序, 不管是否需要得到执行之后的结果的时候, 使用 `execFile` 方法确实是一个不错的选择。比如说, 如果现在需要执行一个图片处理的操作, 并且只关心它是否运行成功, 那么你只需要使用 `execFile` 执行一下 `ImageMagick` 就可以。如果你想执行的这个外部应用会有很多的输出, 并且你又想对输出的结果进行一个实时性分析, 那么这个时候, 使用流则会是一个更好的办法。

技巧 57 流和外部应用程序

设想有这么一个 web 应用程序, 它需要使用一个外部程序的输出。当外部程序输出的数据可用时, 此时你可以选择马上将数据输出到客户端, 通过流, 你可以使用正通过子进程输出的数据。相反, 而不是等到将所有数据缓存好之后再将其输出, 这样一来, 对于调用一个你可能预期有大数据量输出的外部应用, 流确实是一个很好的选择。为什么这样说呢, 因为大量的数据缓存必定占据大量的内存。另一方面, 使用流, 也会提高数据的响应效率, 因为只要一旦有可用的数据, 这些数据马上就可以被使用。

问题

你想通过执行一个外部程序并且得到相应的输出流。

解决方案

通过使用 `spawn` (见图 8.4)。

异步执行我们的外部应用

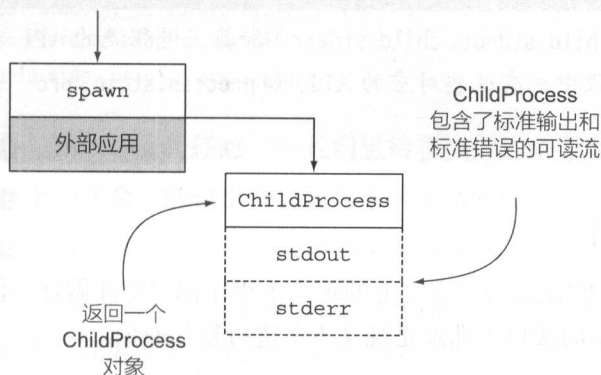


图 8.4 `spawn` 方法返回一个 I/O 的流接口

讨论

spawn 方法有着和 execFile 很相似的方法签名：

```
cp.execFile('echo', ['hello', 'world'], ...);  
cp.spawn('echo', ['hello', 'world'], ...);
```

可以看到，第一个参数仍然是需要被执行的外部程序名，第二个参数就是需要执行的程序的参数，以数组形式提供。和 execFile 不一样，通过返回一个回调方法来提供其缓存好的输出，spawn 方法依赖的是流。

```
var cp = require('child_process');
```

```
var child = cp.spawn('echo', ['hello', 'world']);  
child.on('error', console.error);  
child.stdout.pipe(process.stdout);  
child.stderr.pipe(process.stderr);
```

❶
❷
❸

❶ 返回的 child 对象包含了 stdin、stdout 和 stderr 流对象。

❷ 通过事件来释放异常信息。

❸ stdout 和 stderr 作为数据流对象可以直接被使用。

因为 spawn 方法是基于流的，所以可以很好地处理大数据输出和处理正读取的数据。除此之外，流还有其他好处，比如，child.stdin 是一个可写流，所以通过它可以任何可读流一并写入。同样，对于 child.stdout 和 child.stderr 这样的可读流，就可以被写入到任何可写流中。

API 的对称性

ChildProcess API (child.stdin、child.stdout、child.stderr) 和其父进程流的 API 具有很好的对称性，因为在父进程中也有这些对应的 API，如 process.stdin、process.stdout、process.stderr。

8.1.3 外部应用程序的串联调用

UNIX 系统设计一个重要哲理就是，它让运行于它之上的每一个程序都只专注做好一件事情，然后各个程序之间再通过通用的接口（那就是纯文本）进行相互通信。

现在就让我们通过一个 Node 的程序来演示这样一个场景，有 3 个简单程序通过使用 spawn 方法来处理文本流，并且其相互交互。在 UNIX 中，cat 命令程序用来读取一个文件，并且输出它的内容，然后再使用 sort 命令程序将前面输出的内容作为其输入，再对

内容进行排序后将其输出，最后 `uniq` 命令程序则接收 `sort` 输出的内容，并将其中有重复的行都删除。图 8.5 展示了该场景。

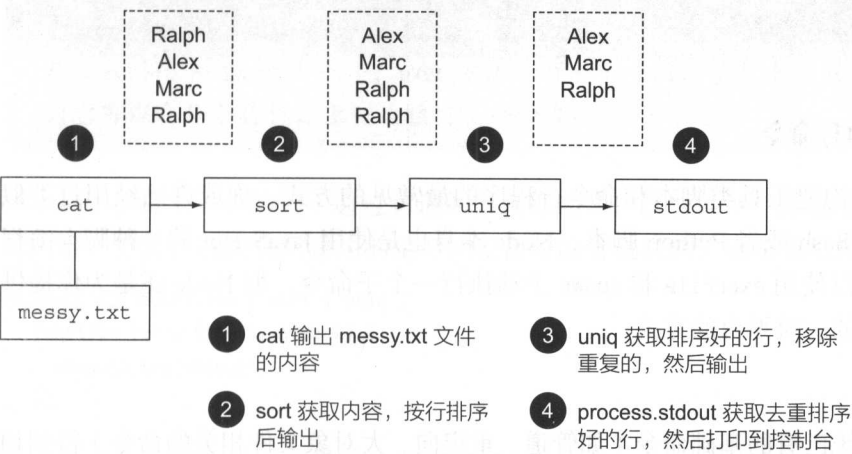


图 8.5 使用 `spawn` 把外部的应用串联起来

具体的代码如下：

```
var cp = require('child_process');
var cat = cp.spawn('cat', ['messy.txt']);
var sort = cp.spawn('sort');
var uniq = cp.spawn('uniq');

cat.stdout.pipe(sort.stdin);
sort.stdout.pipe(uniq.stdin);
uniq.stdout.pipe(process.stdout);
```

- 1
- 2
- 3

- 1 使用 `spawn` 将任意一个我们想接着执行的命令串联起来。
- 2 上一次命令的输出将会成为下一个命令的输入。
- 3 最后将流的结果都写入到 `process.stdout`。

当 Node 处理任意流对象时，使用 `spawn` 方法的流接口确实是一种无缝的处理方式，这也包括处理诸如上述把所有的外部程序串连在一起执行，但是有时候或许通过直接一次性执行一组外部应用程序的命令会来得更加灵活方便，这个时候，就可以考虑使用 `exec`。

应用你所学到的知识

你可以利用在第 6 章所学到的关于 `fs` 模块和流模块的知识，寻找到一种避免使用 `cat` 程序的方案吗？

技巧 58 在 shell 中执行命令

Shell 编程是一种构建工具类脚本和命令行程序的最常见的方式。你或许已经用过类似的脚本语言，如 Bash 或者 Python 脚本，Node 本身也是使用 JavaScript 的一种脚本编程语言。尽管你可以使用 `execFile` 和 `spawn` 手动执行一个子命令，但 Node 还是为你提供了一种更加方便的、跨平台的方式。

问题

你想通过一些系统已有的基础命令（如管道、重定向、大对象文件相关的命令）得到相应的结果。

解决方案

通过使用 `exec`（见图 8.6）。

在子 shell 中异步执行我们的命令

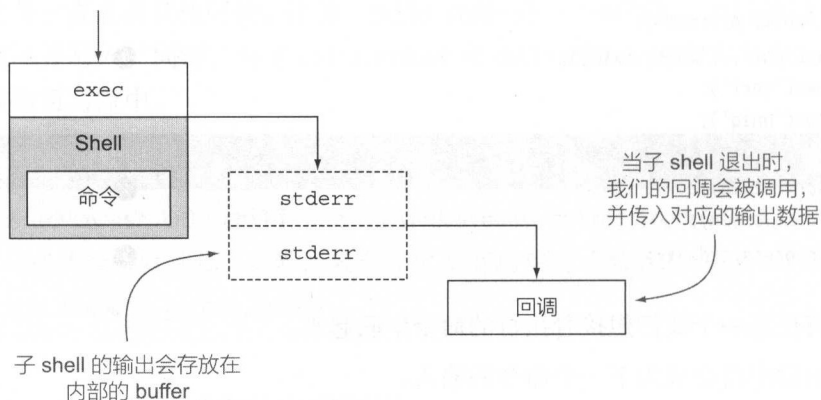


图 8.6 `exec` 命令在一个子 shell 里运行我们的命令

讨论

如果需要在命令解析器里执行命令，你可以选择使用 `exec`。`exec` 方法实际上也是调用 `/bin/sh`（在 UNIX/Linux 下）或者 `cmd.exe`（在 Windows 下）来执行命令的。当然，这

种方法可行的前提是，你必须拥有其他需要被执行的命令的权限（如管道、重定向和后台命令）。

只有一个命令行作为参数

与 `execFile` 和 `spawn` 不一样，`exec` 方法没有一个专门的参数来设置命令名称，因为它可以在命令解析器上一次执行多个命令。

举个例子，将上一节的例子通过管道的方式串起来，以生成一个排过序的没有重复名字的内容。但是，这次我们要用的是 UNIX 自身的管道机制，而不是使用流的方式。

```
cp.exec('cat messy.txt | sort | uniq',  
  function (err, stdout, stderr) {  
    console.log(stdout);  
  });
```

①

②

① 就像我们在命令行中一样，使用管道把 `cat`、`sort`、`uniq` 命令串连在一起。

② 如果成功，输出会包括排序好且不重复的 `messy.txt` 版本。

关于命令解析器

如果你是 UNIX 用户，那么时刻要注意一点，Node 始终是使用 `/bin/sh` 来执行命令的，在现代操作系统中通常是使用 `Bash` 命令解析器，当然也可以选择其他你喜欢的命令解析器。对于 Windows 用户，你可以使用管道的功能，具体使用技巧 57 里讨论的 `spawn` 以流的方式来实现。

8.1.4 安全性和 shell 命令执行

拥有对命令解析器访问的权限之后，你就可以方便地做很多操作。然而这个时候，你也必须要对所做的操作严加谨慎，特别是对你的用户输入行为多加小心。

比如说，如果使用 `xmllint` (<http://xmlsoft.org/xmllint.html>) 来解析和检测用户上传的 XML 文件的格式错误，用户可以上传一个需要用于验证的 xml，诸如下面的代码所示：

```
cp.exec('xmllint --schema '+req.query.schema+' the.xml');
```

如果用户提供的输入数据是“`http://site.com/schema.xsd`”，那么，代入上式代码中，容易得到这样的代码：

```
xmllint --schema http://site.com/schema.xsd the.xml
```

从上述代码中可以看到，输入的参数其实是可以让用户自己提供的，这样一来，这里的参数很容易注入具有非法攻击的命令字符（<https://golemtechnologies.com/articles/shell-injection>）。比如，如果用户输入的是这样的字符信息：“;rm -rf /;”，这将会导致我们运行如下命令（千万不要在你自己的终端运行这行命令！）：

```
xmllint --schema ; rm -rf / ; the.xml
```

或许你还没有发现这里面的猫腻，说明白点：上面这行命令首先执行一个新命令以“;”结束，然后强制性删除根目录下所有的文件（rm -rf /），最后以“;”结束命令。

换句话说，这样的一个输入，它会潜在地删除系统里所有 Node 进程上具有访问权限的文件！而且这只是其中一个可以运行的命令，你系统里所有的文件、命令等，只要你的用户进程拥有权限，都会暴露出来，其带来的风险是不言而喻的。

然而，如果使用类似下面这样的使用 `execFile` 的方式运行外部应用程序，它不会用到命令解析器的功能，这样的方式，就会更加安全。

```
cp.execFile('xmllint', ['--schema', req.query.schema, 'the.xml']);
```

在这种情况下，上面所用的恶意注入是无法执行成功的，因为它并不是通过命令解析器去运行的，这样恶意注入的代码参数将无法作为外部程序的正确参数，从而程序会抛出相应的错误异常。

技巧 59 分离子进程

我们可以通过 Node 打开一个外部应用程序，并且让它单独运行。譬如说，如果在你的 Node 应用里有一个 web 的管理应用程序，它可以为你打开一个长时间地持续运行的进程用于云端做同步工作。设想如果你的 Node 应用程序崩溃了，那么同步进程也会被挂起。为了避免这种情况，你就可以将外部程序分离出来，以使其不受影响。

问题

通过使用 Node 调用一个长时间需要运行的外部应用程序，并且要让它子在子进程在保持运行的状态下能够退出。

解决方案

分离一个衍生（spawned）的子进程（见图 8.7）。

讨论

正常情况下，当父进程终结的时候，所有的子进程都会被终结。子进程被认为是附加到父进程上的。但是 `spawn` 方法可以做到分离一个子进程，从而使得子进程拥有和父进程

一样的级别，即成为一个进程组的头。在这样的场景下，即使父进程终结，子进程也会继续执行直到它自己终结。

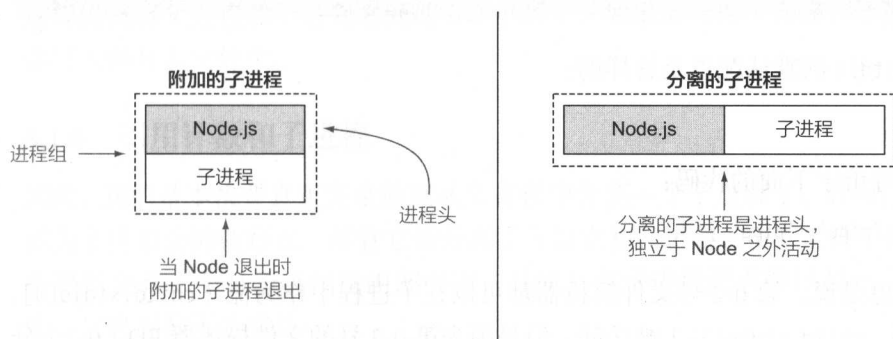


图 8.7 将子进程分离使之独立存在

另外，当你想通过使用 Node 运行一个长时间需要运行的外部进程，并且还希望开始运行之后，Node 就不需要照管它了，这种情况下，上面所讲的分离子进程的场景是十分有用的。

具体的实现方法是使用 `detached` 配置选项，它是 `spawn` 方法的第三个参数，如下面的代码所示：

```
var child = cp.spawn('./longrun', [], { detached: true });
```

在上面的这个代码示例中，`longrun` 将会被提升为一个进程组的头，如果你通过强制性使用 (Ctrl-C) 去终结这个正在运行的 Node 程序，`longrun` 会继续执行，直到它自己终结。

如果不强制性终结正在运行的 Node 程序，就会发现父进程会一直保持活跃状态，直到子进程结束。这是因为子进程的 I/O 和父进程是相互连接的。为了中断这个 I/O 连接，你必须配置另外一个选项：`stdio`。

8.1.5 父进程和子进程之间的 I/O 处理

`stdio` 选项定义了子进程的 I/O 连接到一个具体的地方。它的值可以是一个字符串也可以是一个数组。然而，字符串的赋值方式只不过是为了记法简便，它最后还是会扩展成为一个数组配置的。

当传入的是数组参数的时候，这个数组的结构顺序其实是有讲究的，它的每个索引位置的值都对应一个特定的子进程的文件描述器所指向的 I/O。

什么是文件描述器？

如果你对文件解析器存在疑惑，可以查阅第 6 章的技巧 40。

默认情况下，stdio 的默认配置是这样的：

```
stdio: 'pipe'
```

如果展开，就等价于下面的代码：

```
stdio: [ 'pipe', 'pipe', 'pipe' ]
```

上述代码的意思是说，第 0-2 号文件解析器都可以在子进程中作为流（`child.stdio[0]`、`child.stdio[1]`、`child.stdio[2]`）被访问。但是因为第 0-2 号文件描述器 FD（0-2）分别对应的是 `stdin`、`stdout` 和 `stderr`，它们同样也等价于这样的三个流：`child.stdin`、`child.stdout` 和 `child.stderr`。

这样一来，`stdio` 的值 `pipe` 就连接了父进程和子进程，因为这些流都是开放的，随时待命被读写。但是，有时候我们希望在退出 Node 进程的时候中断这样的一种连接。这里有一种较为粗暴的做法就是简单地将所有创建了的流都用下面这样的代码销毁掉。

```
child.stdin.destroy();
child.stdout.destroy();
child.stderr.destroy();
```

尽管这样做可以达到目的，但是我们并没有意图使用这些创建出来的流，那么最好也就不要一开始就创建这些流。所以另外一种更好的做法是：给某一个文件描述器进行赋值将 I/O 指向别的地方，或者直接用 `ignore` 放弃这个流。

综合上述所说的两种选择，`ignore` 掉 FD0（`stdin`），因为我们并不打算为子进程提供任何输入。以防后期需要做一些调试工作，我们又想从 FDs1（`stdout`）和 FDs2（`stderr`）获得输出。下面是具体的实现：

```
var fs = require('fs');
var cp = require('child_process');

var outFd = fs.openSync('./longrun.out', 'a');
var errFd = fs.openSync('./longrun.err', 'a');

var child = cp.spawn('./longrun', [], {
  detached: true,
  stdio: [ 'ignore', outFd, errFd ]
});
```

①

②

❶ 打开两个日志文件，用于 `stdout` 和 `stderr` 做流的定向。

❷ 第 0 号 FD 直接被忽略了，第二个和第三个参数就是上面定义的两个 log 文件。

这样做就将父进程和子进程之间的 I/O 中断了，如果运行这个程序，子进程的输出会全部写入到日志文件中。

8.1.6 引用计数和子进程

到此，我们基本快要真正完全做到从父进程中分离一个子进程了。然而，其实这还不够，因为子进程会继续存在，尽管它被分离了并且它和父进程的 I/O 也被中断了，但是父进程仍然会有一个对子进程的内部引用，并且只要子进程没有终结且这个引用没有被移除，父进程都不会终结。

可以通过用 `child.unref()` 方法告诉 Node 不要将子进程的引用进行计数。在下面的代码中，整个程序将会在子进程执行 `spawn` 方法之后退出。

```
var fs = require('fs');
var cp = require('child_process');

var outFd = fs.openSync('./longrun.out', 'a');
var errFd = fs.openSync('./longrun.err', 'a');

var child = cp.spawn('./longrun', [], {
  detached: true,
  stdio: [ 'ignore', outFd, errFd ]
});

child.unref();
```

❶ 移除子进程在父进程中的引用。

总结一下，分离一个进程必须做到以下三点：

- 必须将 `detached` 选项设置成 `true`，这样才可以使子进程升级成为它自己的进程组头。
- 必须配置 `stdio` 选项，这样父进程和子进程的 I/O 连接才会中断。
- 父进程中对子进程的内部引用计数必须被移除，可以通过调用 `child.unref()` 来实现。

8.2 执行 Node 程序

诚然，前面讲到的所有技术都可以用来执行 Node 程序，然而，在下面的内容中，我们将会更加专注于如何更有效率地执行 Node 程序。

技巧 60 执行 Node 程序

当使用 Node 编写 shell 脚本、公共工具或者其他命令行应用程序时，易用性和便捷性使得创建一些可执行文件出来是一件很轻松的事情。使用 npm 来发布命令行应用程序也是十分简单方便的。

问题

你想只做一个可以直接执行 Node 程序的脚本。

解决方案

在你当下的平台上创建可执行的文件。

讨论

根据前面讨论过的方法，我们可以用很多种方式使一个 Node 程序可以以子进程的方式来运行，比如简单地调用 node 的 `execFile` 方法：

```
var cp = require('child_process');
cp.execFile('node', ['myapp.js', 'myarg1', 'myarg2'], ...
```

但是很多时候，直接用一個独立的可执行程序会更加方便，比如你可以这样写：

```
myapp myarg1 myarg2
```

在 Windows 和 UNIX 平台下做一个可执行文件的流程各不相同。

在 Windows 平台下做可执行文件

比如说我们现在有一个现成的 `hello.js` 程序，它可以接受下面这样的一个参数：

```
console.log('hello', process.argv[2]);
```

我们可以在命令行里这样运行上面的代码，并得到一个结果：

```
$ node hello.js marty
hello marty
```

为了在 Windows 平台下制作可执行文件，我们可以制作一个批处理文件去调用这个 Node 程序。为了和程序名称一致，我们将这个批处理文件命名成 `hello.bat`，并输入如下内容：

```
@echo off
node "hello.js" %*
```

❶

❷

❶ 不要将正在执行的命令在显示屏上显示出来。

❷ (%*) 正是我们 `hello.js` 代码中需要接受的参数。

现在我们可以直接通过上面的 `hello.bat` 批处理文件来执行 `hello.js` 文件里的程序逻辑。调用方式如下：

```
$ hello tom
hello tom
```

我们可以用子进程的方式来运行这个 `.bat` 可执行文件，如下面的调用：

```
var cp = require('child_process');
cp.execFile('hello.bat', ['billy'], function (err, stdout) {
  console.log(stdout); // hello billy
});
```

在 UNIX 平台下只做可执行文件

为了将一个 Node 程序制作成一个可以在 UNIX 下可执行的程序，不需要像在 Windows 平台下一样使用一个单独的批处理文件，我们可以直接修改 `hello.js`，通过在该文件的最上面增加一段代码，最后修改后的代码如下所示：

```
#!/usr/bin/env node
console.log('hello', process.argv[2]);
```

❶ 执行在用户环境变量中找到的 Node 命令。

然后，为了让程序可以被执行，我们还需要运行下面的命令：

```
$ chmod +x hello.js
```

最后可以在 UNIX 的终端这样直接运行程序：

```
$ ./hello.js jim
hello jim
```

当然，我们也可以对文件进行重命名，让它看起来更加像一个独立的文件：

```
$ mv hello.js hello
$ ./hello jane
hello jane
```

同样，下面的代码展示了以子进程的方式运行这样的—个独立文件：

```
var cp = require('child_process');
cp.execFile('./hello', ['bono'], function (err, stdout) {
  console.log(stdout); // hello bono
});
```

在 NPM 中发布可执行文件

利用 UNIX 的一些约定，可以发布包含可执行文件的包文件，在 Windows 平台下，NPM 则会相应做一些调整。

技巧 61 Forking Node 模块

在处理大数据量计算的任务方面，HTML 里的 web worker (<http://mng.bz/UG63>) 为浏览器和 JavaScript 提供了一种较为优雅的方式。它使得任务脱离了主线程，通过使用一种内置于父进程和子进程间流通信来处理问题。这就降低了大数据量运行的压力，也提供了更好的用户体验。在 Node 中，我们也有类似的概念，那就是使用与之稍有不同的 fork，这种方法同样能帮我们将大批量的任务压力分解到一个独立的进程中，也使得事件轮询能够仍然保持顺畅。

问题

你想操作一个独立的 Node 进程。

解决方案

使用 fork，见图 8.8。

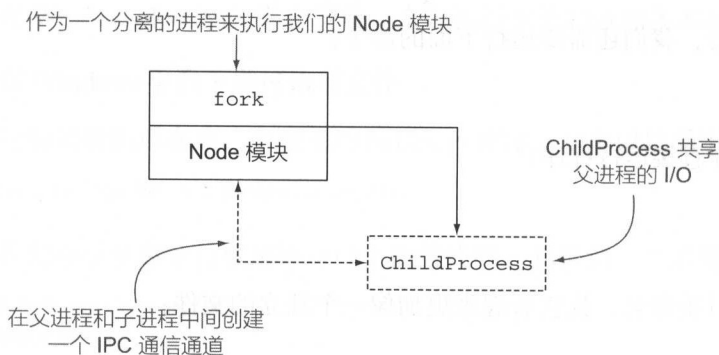


图 8.8 fork 命令通过在一个独立的进程中运行一个 Node 模块，并建立一个通信通道

讨论

有时候使用独立的进程是很有用的，常见的场景就是处理计算任务。因为 Node 本身是单线程的。计算任务直接影响着整个进程的性能。有时候这是可以接受的，但只是对于网络编程而已，当整个进程资源耗尽的时候，这会严重影响请求的效率。在这种情况下，

使用（分叉）forked 好的进程来运行这样的任务就可以让你的应用程序保持很好的响应性。另外一种使用 fork 的用途就是为了共享文件描述器，这种情况发生在当一个子进程需要接受一个父进程传入的连接时。

Node 为其他的 Node 程序间通信提供了一种更好的方式，就是通过设置下面的 `stdio` 配置将它们连接起来。

```
stdio: [ 0, 1, 2, 'ipc' ]
```

这就是说，默认情况下，所有的输入输出都是继承自父进程的，并不会 `child.stdin`、`child.stdout` 或者 `child.stderr`。

```
var cp = require('child_process');
var child = cp.fork('./myChild');
```

如果你想提供一种和 `spawn` 具有一样的默认的 I/O 配置（就是说你想得到一个诸如 `child.stdin` 这样的对象），那么你可以使用 `silent` 选项。

```
var cp = require('child_process');
var child = cp.fork('./myChild', { silent: true });
```

内部进程间通信

虽然有很多种现有的方式可以提供进程通信（IPC；参考：<http://mng.bz/LGKD>），但 Node 的 IPC 通道使用的要么是 UNIX 的域套接字（<http://mng.bz/1189>），要么是 Windows 下的命名管道（<http://mng.bz/262Q>）。

和分叉的（forked）Node 模块进行通信

使用 `fork` 方法会开放一个 IPC 通道，使得不同的 Node 进程间进行消息传送。在子进程这边，它会暴露出 `process.on('message')` 和 `process.send()` 的机制用来接收和发送消息。在父进程这边，则使用的是 `child.on('message')` 和 `child.send()`。

下面举一个简单的例子来具体展示子进程将父进程传入的消息发送给父进程。

```
process.on('message', function (msg) {
  process.send(msg);
});
```

❶ 当子进程收到消息时，这个回调会被调用。

❷ 将消息发送回给父进程。

这样其他的程序通过 `fork` 来使用这个模块，如下：

```
var cp = require('child_process');
var child = cp.fork('./child');
child.on('message', function (msg) {
  console.log('got a message from child', msg);
});
child.send('sending a string');
```

❶

❷

❸

❶ 当父进程收到消息时候，这个回调会被调用。

❷ 把输出的消息打印出来。

❸ 发送一个消息给子进程。

进程间发送的数据类型信息不会丢失，比如说，如果你发送任何一个合法的 JSON 值，它的类型仍然是一个 JSON 格式的。

```
child.send(230);
child.send('a string');
child.send(true);
child.send(null);
child.send({ an: 'object' });
```

从分叉（forked）的 Node 模块中断开连接

因为我们打开了一个父进程和子进程间的一个 IPC 通道，只要子进程不中断，父进程也会保持活动状态。如果需要中断 IPC 通道连接，可以在父进程中这样显式地来实现：

```
child.disconnect();
```

技巧 62 运行作业

当你需要运行一个例行的计算作业时，按需分叉（forking）进程将会耗尽你的 CPU 资源。最好的办法就是构建一个可用的作业池，在池中可以存有足够多的进程并且可以随时等待分配使用。下面就让我们来看看这种技术。

问题

你有一些需要例行工作的作业，而你又不想在主体的事件轮询（Event Loop）中运行这些作业。

解决方案

使用 fork 并管理一个工作池，池中拥有随时待命的可工作的进程。

讨论

我们可以创建一种模式去处理计算量比较大的作业，即通过 `fork` 内置的 IPC 通道。这种技术基于上一个技巧，同时也增加了一个重要的约束，当父进程发送一个任务给子进程的时候，它期望收到一个确切的结果，下面的代码展示了在父进程中这种模式是怎么工作的：

```
function doWork (job, cb) {  
  var child = cp.fork('./worker');  
  child.send(job);  
  child.once('message', function (result) {  
    cb(null, result);  
  });  
}
```

❶

❷

❶ 发送一个作业给子进程。

❷ 期望子进程返回一个确切的消息。

但是收到返回消息只是其中的一种结果，为了让我们的 `doWork` 方法构建得更加有弹性，我们需要考虑以下两点：

- 子进程以任何一种可能的原因退出。
- 意想不到的错误（比如 IPC 通道关闭了，或者是 `fork` 的时候失败了）。

为了解决这些问题，我们必须引入一些监听事件：

```
child.once('error', function (err) {  
  cb(err);  
  child.kill();  
});  
child.once('exit', function (code, signal) {  
  cb(new Error('Child exited with code: ' + code));  
});
```

❶

❶ 异常错误，在不可用的时候杀死进程。

至此，代码看上去已经不错了，但是这里仍然存在多次调用回调函数的风险，比如说工作者进程完成了工作然后退出了或者遇到了一个错误。为了避免这样的问题，我们再优化一下代码，如下：

```
function doWork (job, cb) {  
  var child = cp.fork('./worker');  
  var cbTriggered = false;  
  
  child  
    .once('error', function (err) {
```

❶

❷

```

    if (!cbTriggered) {
      cb(err);
      cbTriggered = true;
    }
    child.kill();
  })
  .once('exit', function (code, signal) {
    if (!cbTriggered)
      cb(new Error('Child exited with code: ' + code));
  })
  .once('message', function (result) {
    cb(null, result);
    cbTriggered = true;
  })
  .send(job);
}

```

- ❶ 跟踪回调是否已经执行。
- ❷ 如果错误发生或者进程退出了，Message 事件永远不会被触发，所以我们不需要检查 cbTriggered。

至此，我们只关注到了父进程。子进程接收了一个作业，并且在结束的时候发送了一个确切的消息返回给了父进程。

```

process.on('message', function (job) {
  // do work
  process.send(result);
});

```

8.2.1 工作池

上面讲到的 doWork 函数在每次做一些工作时都会再创建一个新的子进程。然而，这个并不是没有任何代价的，就像 Node 的官方文档所述：

这些子节点仍然是一个 V8 的新实例。预计每一个节点需要耗费 30 毫秒的启动时间和 10MB 的内存。也就是说，你不能创建太多了，因为这些并不是没有代价开销的。

在处理一些需要消耗很大代价用于处理大量数据计算的任务时，一种性能更好的方式就是：不要每次都打开一个新的子进程，相反，我们可以维护一个工作池，池中都存放了一些可以长时间运行的进程。

现在让我们对 `doWork` 函数进行一下扩展，通过创建一个模块来处理一个工作池。这里我们需要增加一些约束：

- 仅仅根据我们机器上的 CPU 数量来（分叉）`fork` 尽可能多的工作进程。
- 确保一个新的工作任务可以拿到一个可用的进程，而不是一个正在使用的进程。
- 当没有工作进程空闲时，维护一个工作队列，当有工作进程时，再排队处理。
- 按需分叉（`fork`）进程。

让我们来看看下面的代码：

```
var cp = require('child_process');
var cpus = require('os').cpus().length; )

module.exports = function (workModule) {
  var awaiting = [];
  var readyPool = [];
  var poolSize = 0;

  return function doWork (job, cb) {
    if (!readyPool.length && poolSize > cpus)
      return awaiting.push([ doWork, job, cb ]);

    var child = readyPool.length
      ? readyPool.shift()
      : (poolSize++, cp.fork(workModule));
    var cbTriggered = false;

    child
      .removeAllListeners()
      .once('error', function (err) {
        if (!cbTriggered) {
          cb(err);
          cbTriggered = true;
        }
        child.kill();
      })
      .once('exit', function () {
        if (!cbTriggered)
          cb(new Error('Child exited with code: ' + code));
        poolSize--;
        var childIdx = readyPool.indexOf(child);
        if (childIdx > -1) readyPool.splice(childIdx, 1);
      })
      .once('message', function (msg) {
```

```

    cb(null, msg);
    cbTriggered = true;
    readyPool.push(child);
    if (awaiting.length) setImmediate.apply(null, awaiting.shift());
  })
  .send(job);
}
}

```

- ❶ 取到 CPU 的数量。
- ❷ 当没有空闲进程时，用来存放任务队列。
- ❸ 存放准备就绪的工作者进程。
- ❹ 存放现有的工作者进程数量。
- ❺ 如果没有空闲的工作者进程，并且达到了我们设置的限制，排队的工作需要延后运行。
- ❻ 取得下一个可用的子进程或者分叉一个新的进程（增加池子大小）。
- ❼ 取出任何一个子进程上的监听，保证每一个子进程每一次只拥有一个监听。
- ❽ 如果子进程由于某种原因退出，保证它在 readPool 里也被移除了。
- ❾ 子进程再次就绪，将其加回 readPool，如果接下来有等待的任务，运行之。

应用你学到的知识

根据池子的需要可能需要应用一些其他的约束限制。比如说，当任务作业失败时需要重试，或者终止长时间运行的作业，那么我们应该在上面的例子里如何实现这种重试功能或者超时处理呢？

8.2.2 使用池模块

设想根据用户请求在我们的服务器上运行一些需要进行拥有大量密集数据计算的任务时，首先，来扩展一下子工作进程来模拟这样一个需要进行密集计算的任务：

```

process.on('message', function (job) {
  for (var i = 0; i < 1000000000; i++);
  process.send('finished: ' + job);
});

```

- ❶ 从父进程中接收任务。
- ❷ 在子进程上模拟现实中的高负荷任务。
- ❸ 将结果发回给父进程。

既然你现在可以运行一个简单的子进程，综合上面讨论的知识，通过下面的示例代码来展示池模块和工作者进程模块。

```
var http = require('http');  
var makePool = require('./pooler');  
var runJob = makePool('./worker');  
  
http.createServer(function (req, res) {  
  runJob('some dummy job', function (er, data) {  
    if (er) return res.end('got an error:' + er.message);  
    res.end(data);  
  });  
}).listen(3000);
```

- ❶ 通过引入池模块来创建工作池。
- ❷ 通过工作者进程模块来创建工作池。
- ❸ 根据每一次的向服务器发送请求来运行作业，并且返回相应的结果。

通过使用池机制，节约了创建和销毁子进程的开销。它通过利用 `fork` 内置的通信通道，允许 Node 有效地通过跨越一组子进程进行管理任务作业。

进一步探索

你通过可以检验第三方模块 `compute-cluster` (<https://github.com/lloyd/node-compute-cluster>) 更深入地了解工作池。

我们已经讨论了子进程的异步执行，通过这个，你可以玩转多点 I/O，比如多个服务器。但是有时候你只是想无开销地一个接一个地执行命令。我们在接下的章节中就会探讨这个问题。

8.3 同步运行

为了使事件轮询持续工作，并且使它无须等待一个难以控制的子进程的结束，无阻塞 I/O 在这方面就显得尤为重要。然而这也需要额外的代码开销，并且有时候你可能就需要一个阻塞时，这样代码会让人感到很很不愉快。举个很好的例子，就是当你编写 shell 脚本的时候遇到这样的问题。幸运的是，我们可以使用同步方式运行的子进程！

技巧 63 同步子进程

同步子进程方法是最近加到 Node 中的。最早在 Node 0.12 版本中，它用来使用一种高性能和类似 shell 编程的方式来解决一个很现实的问题。在 0.12 版本之前，这种看似同步

的行为常常被一些聪明的不良黑客所使用。现在，这种方法也被认为是一种公然认可的方法了。

在这个技巧中，我们将会讨论所有的可用于同步子进程模块的方法。

问题

你想同步执行一些命令。

解决方案

使用 `execFileSync`、`spawnSync` 和 `execFile`。

讨论

至此，我们希望所有的同步方法看上去都相似，事实上，它们不管是在方法签名还是在目的上都和这章之前讨论的方法上，具有很多的相似点。唯一不同的一点就是，它们在被调用的时候需要阻塞并且一直到运行结束。

如果你只想同步执行一个单独的命令，并且得到一个输出，那么可以使用 `execFileSync`：

```
var ex = require('child_process').execFileSync;
var stdout = ex('echo', ['hello']).toString();
console.log(stdout);
```

❶
❷
❸

- ❶ 使用 `ex` 引用 `execFileSync`，方便记忆。
- ❷ 使用 `ex` 引用 `execFileSync` 返回一个输出的缓存，并且最后转换成一个 UTF-8 的字符串赋给 `stdout`。
- ❸ 输出 “hello”。

如果你想程序式地同步执行多个命令，并且命令之间的结果存在相互依赖的关系，这种情况下，可以用 `spawnSync`：

```
var sp = require('child_process').spawnSync;
var ps = sp('ps', ['aux']);
var grep = sp('grep', ['node'], {
  input: ps.stdout,
  encoding: 'utf8'
});
console.log(grep);
```

❶
❷
❸
❹

- ❶ 使用 `sp` 引用 `execFileSync`，方便记忆。
- ❷ 运行 `ps aux`，并且同步运行 `grep node`。
- ❸ 将从 `ps aux` 得到 `stdout` 的缓冲作为 `grep node` 的输入。

④ 表明所有的结果 `stdio` 应该是 UTF-8 格式的。

同步子进程得到的结果包含了很多细节，这也是使用 `spawnSync` 的另外一个好处。

```
{ status: 0, ①
  signal: null, ②
  output: ③
    [ null,
      'wavded 4376 ... 9:03PM 0:00.00 (node)\n
        wavded 4400 ... 9:11PM 0:00.10 node spawnSync.js\n',
      '' ],
  pid: 4403, ④
  stdout: 'wavded ... 9:03PM 0:00.00 (node)\n
    wavded 4400 ... 9:11PM 0:00.10 node spawnSync.js\n', ⑤
  stderr: '', ⑥
  envPairs: ⑦
    [ 'USER=wavded',
      'EDITOR=vim',
      'NODE_PATH=/Users/wavded/.nvm/v0.11.12/lib/node_modules:',
      ... ],
  options: ⑧
    { input: <Buffer 55 53 45 52 20 20 20 ... >,
      encoding: 'utf8',
      file: 'grep',
      args: [ 'grep', 'node' ],
      stdio: [ [Object], [Object], [Object] ] },
      args: [ 'grep', 'node' ], ⑨
      file: 'grep' } ] ⑩
```

① 进程的退出状态。

② 用来结束进程的信号量。

③ `stdio` 流的所有输出，这些流在子进程中会被用到，我们可以看到 1 号参数位置（`stdout`）就有数据。

④ 进程编号。

⑤ 进程的 `stdout` 输出。

⑥ 进程的 `stderr` 错误。

⑦ 程序运行时的当前环境变量。

⑧ 用来创建进程的选项；从这里我们可以看到来自于 `ps aux` 的输入缓存。

⑨ 用于执行进程的参数。

⑩ 执行文件。

最后，还有一个 `execSync` 方法，它可以同步地执行一个子命令。在用 JavaScript 编写 shell 脚本的时候这个是很便利的。

```
var ex = require('child_process').execSync;  
var stdout = ex('ps aux | grep').toString();  
console.log(stdout);
```

❶

❷

❶ 使用 `ex` 引用 `execSync`，方便记忆。

❷ 同步地执行 shell 命令并且返回一个字符串输出。

上面的代码将得到如下所示的输出：

```
wavded 4425 29.7 0.2 ... 0:00.10 node execSync.js  
wavded 4427 1.5 0.0 ... /bin/sh -c ps aux | grep node  
wavded 4429 0.5 0.0 ... grep node  
wavded 4376 0.0 0.0 ... (node)
```

❶

❶ 这个输出显示我们正在使用 `execSync` 执行一个子 shell 命令。

同步子进程中的异常处理

如果在 `execSync` 或 `execFileSync` 执行的结果中返回的是一个非零状态，这种情况下，将会有异常抛出。这个抛出的异常对象将会包含我们在使用 `spawnExec` 返回的结果里的所有东西。我们可以访问状态编码里的重要信息和 `stderr` 流。

```
var ex = require('child-process').execFileSync;  
try {  
  ex('cd', ['non-existent-dir'], {  
    encoding: 'utf8'  
  });  
} catch (err) {  
  console.error('exit status was', err.status);  
  console.error('stderr', err.stderr);  
}
```

❶

❷

❶ 执行一个不存在的目录，将会返回一个非零状态。

❷ 虽然比 `toString()` 方法更啰唆，但当我们在处理异常的时候还是将所有 `stdio` 流结果都设置成 UTF-8 格式。

这个程序的输出将会是这样的：

```
exit status was 1  
stderr /usr/bin/cd: line 4:cd:  
non-existent-dir: No such file or directory
```

在讲 `execFile` 和 `execFileSync` 的时候提过异常，那对于 `spawnSync` 又是怎样的呢？因为 `spawnSync` 返回的所有东西都是在运行进程时返回的。所以，它不会抛出异常，因此，你有必要去检查一下它成功与否。

8.4 总结

在本章，我们学习了在 Node 中通过使用 `child_process` 模块来整合外部程序的不同方法，下面总结一些建议：

- 何时使用 `execFile`？当你只需要执行一个外部程序的时候使用它。这种方法的执行速度快，使用简便，并且在处理有用户输入时相对更加安全。
- 何时使用 `spawn`？当你想处理一些会有很多子进程 I/O 的事情时，或者当你期望的进程会有大量的输出时，这种方法提供了一个很好的流式接口，同样在处理有用户输入的时候相对更加安全。
- 何时使用 `exec`？当你想直接访问一些现成的 shell 命令的时候，并且希望很多的 shell 命令允许多个命令一次性运行。但是使用这种方法就一定要注意用户输入了，如果将不受信任的输入数据作为命令的输入就是一种很不明智的选择。
- 何时使用 `fork`？当你想将一个 Node 进程作为一个独立的进程来运行的时候，这种方法会使得计算处理和文件描述器脱离 Node 的主进程。
- 分离那些你想在即使 Node 进程终止了还想保持活动的衍生（`spawned`）进程。这允许我们使用 Node 启动长时间运行的进程，并且它们可以独立运行。
- 将一组 Node 进程放在池中进行管理，并且使用内置的 IPC 通道。这样做节约了在 `fork` 的时候新建和销毁一个进程的开销。对于构建用于计算业务的 Node 进程集群方面也十分有用。

到这章为止，我们基本讲解完了 Node 的一些基础知识，在前面的章节里主要专注于核心模块的功能讲解和 Node 的惯用原则方法。在下面的章节里，我们将会对核心的概念加以扩展并更加关注实际的开发运用。

[illegible]

48

1. 根据《公司法》第 151 条规定，股东有权请求法院撤销股东会决议，但须以决议内容违反法律、行政法规强制性规定为前提。本案中，股东会决议内容并未违反法律、行政法规强制性规定，故原告请求撤销该决议，缺乏法律依据，不予支持。

有文章「[The Power of the Word](#)」中談到，在英語中，*show* 和 *display* 都指「展示」或「配合」，
而 *show* 更強調「展示」的動作，而 *display* 則強調「展示」的結果。

第二部分

实践中的技巧

在这本书的第一部分，我们深入了解了 Node 的标准类库。现在我们将更多地来看下在很多 Node 程序实践中出现的技巧。Node 以编写快速的网络应用（高性能的 HTTP 解析、Express 之类的易用框架）而著称，所以我们投入了一整章的内容用来讲述 web 开发。

同时，有一些章节来帮助你理解 Node 程序是如何提前做测试以及出现问题后如何调试的。总之，我们是想要让你可以成功地把应用发布到生产环境。

9 网络：构建精简的网络应用

本章概要

- 使用 Node 开发客户端
- 浏览器的 Node
- 服务端技术和 WebSockets
- 应用从 Express 3 迁移到 Express 4
- 测试 web 应用
- 全栈框架和实时服务

本章的目的是把你已经了解的网络、缓冲区、流和测试各部分的知识汇集起来，来编写更好的 web 应用程序。主要的内容包括了基于浏览器的 JavaScript、服务器端代码和测试实用技术等。

无论你的背景情况是怎样的，Node 都可以帮助你写出更好的 web 应用程序。如果你是一个客户端的开发者，那么你会发现它可以帮助你更有效地工作。你可以使用它进行客户端静态资源的预处理以及管理客户端的工作流程。如果你想快速地搭建起一个 HTTP 服务器，来为一个单页的 web 应用，或者一个网络来构建需要的 CSS 或者 CoffeeScript，那么 Node 是一个绝佳的选择。

这个系列的上一本书, *Node.js in Action*, 对使用 Connect 和 Express, 以及使用 Jade 和 EJS 这些模板语言进行 web 开发有了一个比较详细的介绍, 如果你是一个 Node 新手, 那么建议你也阅读一下 *Node.js in Action*; 如果你是已经在使用 Express 了, 那么希望你可以在这一章内容中了解到更多东西; 我们会讲解到构建 Express 应用的相关技术, 并且让其更加容易扩展来适应项目的成长。

这一章的第一部分内容主要关注浏览器部分。如果你是一个曾经因为客户端类库需要而使用过 Node, 且带有困惑的前端工程师, 那么你应该从这一部分开始。如果你是一个服务端的开发者, 想要把 Node 带到浏览器中, 那么可以直接到技巧 66 中查看如何在浏览器中使用 Node 模块。

9.1 前端技术

这一节内容主要讲述 Node 和客户端方面相关的技术。你可以看到在 Node 中如何使用 DOM, 以及如何运行你本地的开发服务器。如果你原本是一个 web 设计者, 也可以看一下 Node 如何自动化处理一些前端的杂事。

第一个技术点将会为你展示如何为简单的网站或者单页面的 web 应用创建一个快速的静态服务器。

技巧 64 快速的静态网站服务器

有时候你仅仅想要一个 web 服务器用来提供静态网站的访问或者单页面的 web 应用, Node 会是一个很好的选择, 因为它可以很简单地运行一个 web 服务器。它也可以很简洁地封装客户端的一个工作流, 来方便和其他人进行协同工作。除了常规的使用客户端的 JavaScript 和 CSS 来运行程序, 你可以编写 Node 程序以便其他人也可以访问。

本节会介绍开启一个 web 服务器的三种解决方案: 一个简短的 Connect 脚本, 一个命令行的 web 服务器, 以及使用 Grunt 的一个小型构建系统。

问题

你需要快速搭建一个服务器来帮助你开发一个静态网站或者一个单页面的应用。

解决方案

使用 Connect, 一个命令行式的 web 服务器, 或者一个类似 Grunt 的客户端工作流。

讨论

过去纯粹的 HTML、JavaScript、CSS 和图片不需要服务器便可以在浏览器中进行访问。

但是因为大多数的 web 开发任务都是从服务器上的某些文件开始的，你经常需要一个服务器来创建一个静态的网站。这是一个麻烦事，但也可以变得更加简单。现在的浏览器已经足够强大，你可以通过访问外部的 web API 来创建复杂的 web 应用，如单页面的 web 应用，或者所谓的无服务端应用。

在很多的所谓无服务端的 web 应用中，你可以使用构建工具类预处理和打包客户端的文件资源，来让工作更加有效率。这一节的内容会展示如何创建一个开发静态网站的 web 服务器以及如何使用如 Grunt 这一类工具来快速简单地开始一个小型项目的开发。

虽然你可以使用 Node 内置的 http 模块来创建静态网站服务器，但是需要一些工作。你需要处理一些问题，例如检测文件的内容类型来发送一个正确的 http 头部等。http 核心模块是坚实基础的一部分，但是你可以使用第三方模块来节省时间。

首先，我们看一下如何使用 Connect，这是流行的 Express 框架中的一个 HTTP 中间件模块，来创建一个 web 服务器。第一个例子是很简单的。

例子 9.1 一个快速的静态 web 服务器

```
var connect = require('connect');
```

```
connect.createServer(  
  connect.static(__dirname)  
).listen(8080);
```

❶
❷
❸

❶ 基于 Node 标准的 HTTP 服务来创建一个 web 服务器。

❷ 提供当前目录的文件访问。

❸ 监听 8080 端口。

要使用图 9.1 中的例子，你需要安装 Connect 模块。你可以运行 `npm install connect`。但是更好的方式是使用一个 `package.json` 文件，以便其他人可以理解你的项目是如何工作的。尽管只是一个简单的静态网站，创建一个 `package.json` 可以在项目成长的未来有效地进行管理。你所需要做的仅仅是运行 `npm init` 和 `npm install --save connect`。第一个命令会给当前目录创建一个信息描述文件，第二个命令会安装 connect 模块，并且将其保存在新创建的 `package.json` 文件中的依赖配置一项。了解了这些之后，你可以很随意地创建新的 Node 项目。

`createServer` 方法❶是从 Node 中的 `http.createServer` 衍生而来的，它被 connect 在背后包装了一些东西。`static` 服务器的中间组件❷是用于提供当前目录的文件访问服务（`__dirname` 前缀是两个下画线的这个变量代表了当前目录），你可以修改成任意你要的

路径。例如,你的客户端资源都存放在 `public/` 目录下,你可以改为使用 `connect.static(__dirname + '/public')`。

最后,服务器被设置为监听 8080 端口^③。这意味着运行这个脚本后,在浏览器访问 `http://localhost:8080/file.html`, 可以看到 `file.html`。

如果已经从设计师那边获取了许多 `html` 文件,由于它们的图片和 `CSS` 文件的路径都以 `/` 开头,你就想使用一个服务器来进行查看。在 `npm` 中有很多模块可以选择使用,它们也支持了不同的开发选项。例如, `Jesse Keane` 的 `glance` 模块,你可以在 `Github` 中找到: `https://github.com/jarofghosts/glance` 或者在 `npm` 中寻找 `glance`。

在命令行中使用一目了然,先到你要查看的 `HTML` 文件所在的目录。然后使用 `npm install --global glance` 来进行全局安装 `glance`。最后输入 `glance`。现在你可以在浏览器中打开 `http://localhost:61403/file`, 来浏览你想要查看的文件。

`glance` 可以使用多种方式进行配置,你可以使用 `--port` 来把监听端口从默认的 61403 改成其他的,可以使用 `--dir` 来执行服务开启的文件目录。输入 `--help` 可以查看配置的列表。它也提供了一些很棒的东西,例如默认的 404 页面,如图 9.1 所示一样。

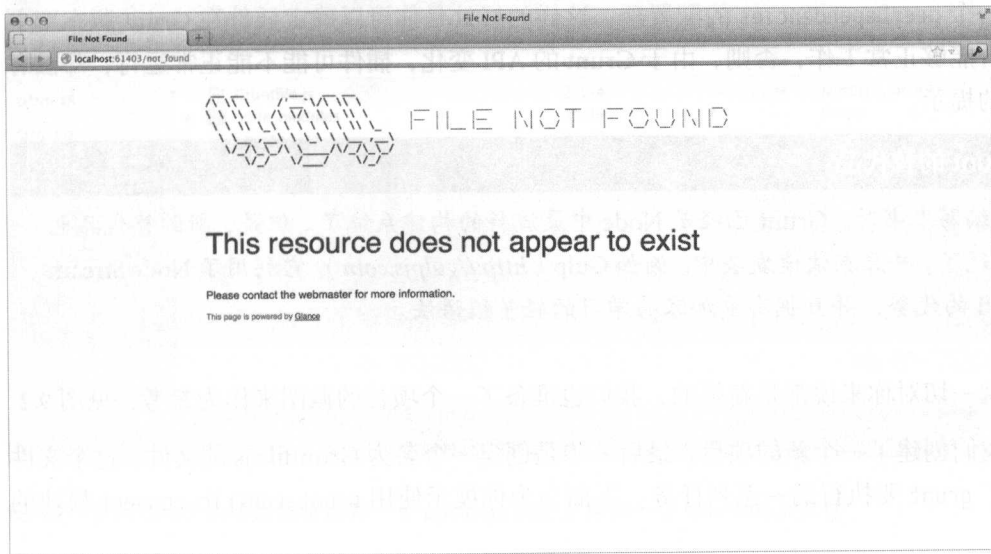


图 9.1 Glance 有内置的错误页面

第三种运行一个 `web` 服务器的方法是使用如 `Grunt` 的任务执行器。这可以进行客户端任务的自动化运行,并且其他人也可以直接使用。使用 `Grunt` 有点像刚才另外两种方式结合在一起,它需要一个像 `Connect` 这样的 `web` 服务模块,以及需要一个命令行工具。

在客户端项目中使用 Grunt，你需要做三件事情：

1. 安装 grunt-cli 模块。
2. 创建一个 package.json 文件来管理你项目的依赖。
3. 使用 Grunt 插件来运行一个 web 服务器。

第一步很简单，使用 `npm install -g grunt-cli` 来全局安装 grunt-cli 模块。现在你可以在任意使用 grunt 项目中输入 grunt 来运行 Grunt 任务了。

下一步，为项目创建一个新的目录。在创建的目录下输入 `npm init`，你可以直接按回车键使用每一项的默认值。现在你需要安装一个 web 服务模块，可以这样操作：`npm install --save-dev grunt grunt-contrib-connect`。

上边的命令同样把 grunt 作为依赖安装了。主要是为了把 Grunt 锁定为当前的版本，如果你打开 package.json，可以看到 "grunt": "~0.4.2"，这意味着首次安装 Grunt 时版本为 0.4.2，将来可以使用在 0.4 分支上更新的版本。一些类似 Grunt 的模块相当流行，这让 npm 不得不支持如同级依赖（peer dependencies）这些特性。同级依赖允许 Grunt 插件指定一个特定版本的 Grunt 作为依赖项，所以我们使用的 Connect 模块在它的 package.json 会有一个 peerDependencies 的配置项。这样的好处是你可以确定插件在 Grunt 版本变更时是否能够正常工作，否则，由于 Grunt 的 API 变化，插件可能不能正常运行，却没有明显的提示。

Grunt 的替代品

在编写本书时，Grunt 已经是 Node 中最流行的构建系统了。但是，新的替代品也出现了，并且在快速发展中。例如 Gulp (<http://gulpjs.com>)，它利用了 Node Stream API 的优势，并且拥有更加容易学习的轻量级语法。

如果这一切对你来说都是新鲜的，我们也准备了一个项目的截图来作为参考，见图 9.2。

现在我们创建了一个新的项目，最后一步是创建一个名为 Gruntfile.js 的文件。这个文件包括了 grunt 要执行的一系列任务。下面会为你展示使用 grunt-contrib-connect 模块的例子：

例子 9.2 一个 Gruntfile 文件来给静态内容提供服务

```
module.exports = function(grunt) {  
  grunt.loadNpmTasks('grunt-contrib-connect');  
  
  grunt.initConfig({
```

①

②

```
connect: {
  server: {
    options: {
      port: 8080,
      base: 'public',
      keepalive: true
    }
  }
}

grunt.registerTask('default', ['connect:server']);
};
```

① 所有的 Gruntfile 都是对外暴露一个方法。

② 加载 Connect 插件。

③ 静态文件的路径。

④ 默认的命令在这里进行重命名。

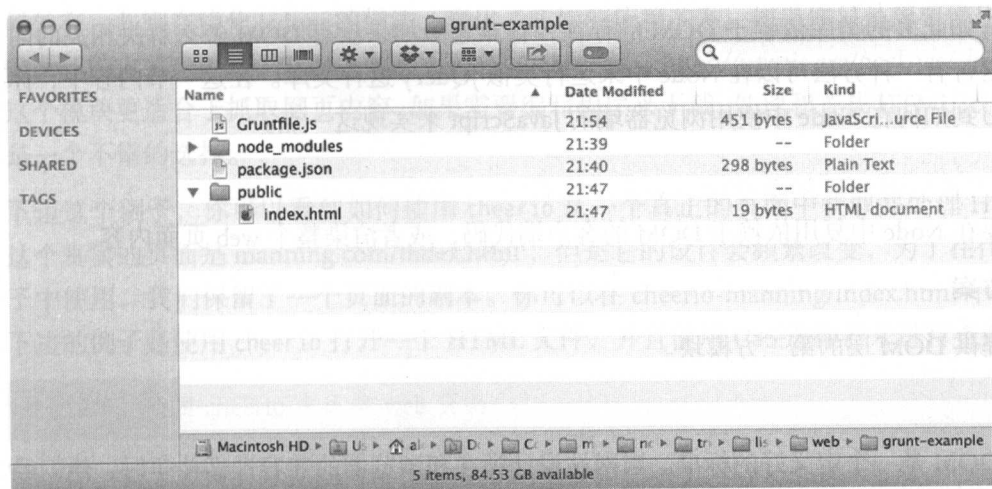


图 9.2 使用 Grunt 的项目通常都有一个 package.json 文件和一个 Gruntfile.js 文件

你也可以创建一个 public 的目录，并且在里边存放一个 index.html 文件，内容可以随意。然后，在已经有了 Gruntfile.js 的目录中运行 `grunt connect`，web 服务便已经启动了。你也可以输入 `grunt` 来运行，因为我们已经把默认的任务设置为 `connect:server` ④。

Gruntfile 使用 Node 标准的模块系统，接收一个 grunt 的对象①来创建任务。插件是使用 `grunt.loadNpmTasks` 进行加载，引用 npm 安装的模块②。大部分插件都有自己的配置选

项，这些都在一个对象中设置后传递给 `grunt.initConfig`，我们已经定义了服务端口和路径，你也可以修改 `base` 属性来进行调整^③。

使用 Grunt 来启动一个 web 服务器，比起使用 glance 会麻烦一点，但是你可以看一下 Grunt 的插件列表（<http://gruntjs.com/plugins>），你会看到有 2000 多个选择，几乎覆盖了前端需要的任务，从 CSS 文件压缩到亚马逊的 S3 云服务的集成等。如果你需要合并客户端的 JavaScript 文件，或者创建图片的 sprites 图，也有对应的插件来进行自动化处理。

下一个技巧你可以了解到如何在 Node 中复用客户端的代码。我们会为你展示在 Node 进程中如何渲染 web 端的内容。

技巧 65 在 Node 中使用 DOM

随着相关工作的开展，在 Node 中模拟浏览器已经是可以实现的了。如果你想要编写 web 爬虫（把 web 页面转换为结构化内容的程序），这是非常有用的。这一部分的技术会比你想象的要复杂一些，因为浏览器不仅仅提供了 JavaScript 运行环境，同时也提供了在 Node 中没有的文档对象模型（DOM）的 API。

已经有如此多的类库依赖于 DOM 的存在，很难想象如果没有 DOM 怎么解决相关的问题。是否有一种方法可以在 Node 中来运行类似 jQuery 这种类库。在这一节内容中，你会学习到如何在 Node 中使用浏览器端的 JavaScript 来实现这一点。

问题

你需要在 Node 中复用依赖于 DOM 的客户端代码，或者渲染整个 web 页面内容。

解决方案

使用提供 DOM 层的第三方模块。

讨论

W3C DOM 是一个定义好的标准。当网站设计师在和浏览器的兼容性做斗争时，他们经常面对的情况是，标准需要一个深入的解释，但是每一个浏览器厂商对标准的理解和实现都略有不同。如果你的目的是运行依赖于 DOM APIs 的 JavaScript，你是幸运的，在 Node 中这些标准被重新梳理了，允许你运行流行的客户端类库。

早期的一个解决方案是 jsdom（<https://github.com/tmpvar/jsdom>）。整个模块接收一个环境的配置，然后提供一个 window 对象。如果使用 `npm install -g jsdom` 安装了，那么你应该可以运行下边这个例子的代码：

```
var jsdom = require('jsdom');
```



```
jsdom.env(  
  '<p class="intro">Welcome to Node in Practice</p>',  
  ['http://code.jquery.com/jquery.js'],  
  function(errors, window) {  
    console.log('Intro:', window.$('intro').text());  
  }  
);
```

❶ 你想要处理的 HTML 代码。

❷ jsdom 要加载的第三方类库。

❸ 现在可以使用 jQuery 的 `$()` 方法了。

这个例子，输入了要处理的 HTML ❶，从远处获取依赖的脚本 ❷，然后提供了一个类似浏览器中的 window 对象 ❸。这已经可以让你使用 jQuery 来处理 HTML 代码片段了，jQuery 就像它在浏览器中一样。这是非常有用的，你可以如同平时一样，编写脚本来处理 HTML 文档，不必使用解析器，你可以使用你熟悉的工具来查询和处理 HTML。这对编写 web 爬虫之类的程序来说简直是极极了，不再那么令人沮丧和乏味。

另外有一些则在优化 jsdom 的做法，简化了底层的依赖关系。如果你只是想要使用类似 jQuery 的方式来处理 HTML，那么可以使用 cheerio (<https://npmjs.org/package/cheerio>)。这个模块更适合于抓取网页内容。如果需要编写程序来下载、处理和查找 HTML，cheerio 是一个不错的选择。

下边这个例子，你可以看到如何使用 cheerio 从一个真正的页面中获取并处理 HTML。这个真实的页面是 manning.com/index.html，但是它的设计会频繁改变，为了在代码例子中使用，我们保留了一个页面的副本。你可以在 `cheerio-manning/index.html` 中找到。下边的例子是使用 cheerio 打开一个 HTML 文件，并且使用 CSS 选择器来进行查询。

例子 9.3 使用 cheerio 来抓取 web 页面

```
var cheerio = require('cheerio');  
var fs = require('fs');  
  
fs.readFile('./index.html', 'utf8', function(err, html) {  
  var $ = cheerio.load(html);  
  var releases = $('Releases a strong');  
  
  releases.each(function(i) {  
    console.log('New release:', this.text());  
  });  
});
```

- ❶ 加载 HTML 内容。
- ❷ 使用 CSS 选择器进行查询。
- ❸ 获取文本内容。

HTML 文件使用 `fs.readFile` 来加载。如果在现实情况中这么做，你需要先使用 HTTP 来下载这个页面文件，你可以使用 `http.get` 来替代 `fs.readFile`，直接从网络上获取 Manning 的首页内容。在第 7 章技巧 51 中有一个关于 `http.get` 的详细例子。

当 HTML 内容获取之后，传递给 `cheerio.load` 方法❶。把结果设置为 `$` 变量，如果使用 jQuery，这样会让代码更加容易理解，但是你也可以命名为其他。

现在所有的东西都设置好了，你可以查询 HTML 内容。`$('.Releases a strong')`❷用于在文档中查询已经发布的最新的书籍。它们在一个 `class` 为 `Releases` 的 `div` 中，是一个 `a` 标签。

每一个元素可以使用 `releases.each` 来进行迭代，类似 jQuery。回调函数的上下文会是当前元素，所以可以使用 `this.text()` 来获取当前元素节点的文本内容❸。

因为 Node 有如此多的第三方模块，你可以利用这个例子做出更多令人惊讶的事情。添加 Redis 缓存和处理的网站队列，然后把结果收集起来，并丢给 Elasticsearch，你便有了自己的搜索引擎！

现在你看到了如何在 Node 上运行浏览器端的 JavaScript，但是相反是怎么样的？你可能需要在客户端复用一些 Node 的代码，或者你可能只是想要使用 Node 的模块系统来组织客户端代码。在下面的技巧中，你可以学到如何在浏览器运行 Node 脚本。

技巧 66 在浏览器端使用 Node 模块

其中 Node 的一个关于 JavaScript 的卖点是你可以服务器上复用已有的浏览器端的编程技能。但是如何在浏览器上不做修改来复用 Node 的代码？这不是很酷么？这里有一个例子：可以在 Node 中定义数据模型，处理类似数据验证的事情，并且你想要在浏览器复用它们，当数据不可用时可以自动展示错误信息。

这几乎是可能的了，但并不是完美的：不幸的是浏览器有很多必须处理的怪毛病。同时，客户端的 JavaScript 也没有 `require` 这个重要的特性。在这个技巧中，你会看到如何把为 Node 编写的代码转化成在大部分 web 浏览器中可以工作的代码。

问题

你想要使用 `require()` 来组织你的客户端代码，或者在浏览器复用 Node 的模块。

解决方案

使用类似 Browserify 的程序来把 Node 的 JavaScript 代码转换为对浏览器友好的代码。

讨论

在这种技术中,我们将使用 Browserify (<http://browserify.org/>) 把 Node 模块转换成对浏览器友好的代码。也存在着其他的解决方案,但在这一点 Browserify 是较成熟的和流行的解决方案之一。它不仅是使用补丁来支持 `require()`, 也可以使用依赖于 Node 的 `stream` 和网络的 API 代码。你甚至可以用它来递归转换从 `npm` 中获取的模块。

它是如何工作的,首先来看一个简短独立的例子吧。在开始时,使用 `npm` 安装 Browserify: `npm install -g browserify`。一旦你已经安装了 Browserify,可以使用 `browserify index.js -o bundle.js` 来将 Node 模块转换为浏览器脚本。任何 `require` 语句都会把对应的文件打包进 `bundle.js`,你不应该更改这个文件。相反地,一旦原来的文件被修改,这个文件就会被覆盖。

例子 9.4 展示了一个使用 `EventEmitter` 和 `utils.inherit` 来创建基础消息类的例子。

例子 9.4 在浏览器使用 node 模块

```
var EventEmitter = require('events').EventEmitter;
var util = require('util');

function MessageBus(options) {
  EventEmitter.call(this, options);
  this.on('message', this.messageReceived.bind(this));
}

util.inherits(MessageBus, EventEmitter);

MessageBus.prototype.messageReceived = function(message) {
  console.log('RX:', message);
};

var messageBus = new MessageBus();
messageBus.emit('message', 'Hello world!');
```

❶ 如同平常一般使用 `require()` 来加载模块。

❷ `EventEmitter` 可以使用 `util.inherits` 来继承。

运行 Browserify 来处理这个脚本会生成一个约 1000 行代码的 `bundle` 文件! 但是我们可以如同在任何 Node 程序中一样使用 `require` ❶, 并且 Node 模块也可以工作, 如例子 9.4

中使用了 `util.inherits` 和 `EventEmitter` ②。

有了 `Browserify`，你也可以使用 `require` 和 `module.exports`，这比使用 `<script>` 标签来处理好多了。前边的例子进行扩展可以做到这一点。在例子 9.5 中，`Browserify` 用于创建一个客户端脚本，它使用 `require` 加载 `MessageBus` 和 `jQuery`，并且在消息触发时修改 DOM。

例子 9.5 在浏览器使用 node 模块

```
var MessageBus = require('./messagebus');
var messageBus = new MessageBus();
var $ = require('jquery')(window);
```

①

```
messageBus.on('message', function(msg) {
  $('#messages').append('<p>' + msg + '</p>');
});
```

```
$(function() {
  messageBus.emit('message', 'Hello from example 2');
});
```

②

① `jQuery` 可以使用 `Browserify` 来加载！

② 可以使用 `jQuery` 的 `DOM ready` 方法。

通过创建一个 `package.json` 把 `jquery` 作为依赖，你可以使用 `Browserify` 来加载 `jQuery` ①。这里我们用它来创建一个 `DOMContentLoaded` 监听器 ②，并且当收到消息时，添加一个段落到容器元素。

Source maps

如果 `Browserify` 生成的 JavaScript 文件抛出错误，那么它可能很难在堆栈中跟踪代码行号，因为它们涉及到源码的行号。如果构建脚本时，携带了 `--debug` 标识，那么 `Browserify` 将生成指向原始文件和行号的映射。

这些映射需要兼容的调试器——你还需要开启浏览器的调试工具来使用它们。在 `Chrome` 中，需要在 `Chrome` 的 `DevTools` 下的选项选择启用 `Source maps`。

为了让这个代码可以工作，所有你需要做的便是在例子 9.4 中，添加 `module.exports = MessageBus`，然后生成 `bundle: browserify index.js -o bundle.js`，其中 `index.js` 的内容是例子 9.5 中的代码。`Browserify` 会忠实地根据 `index.js` 文件中 `require` 语言，从 `./node_modules` 获取 `jQuery` 和从 `messagebus.js` 中获取 `MessageBus` 类。

因为人们可能会忘记如何构建脚本，可以添加一个脚本，放到 `package.json` 文件，如：
"build": "browserify index.js -o bundle.js"。本书中可以下载的代码示例包括了一个 `package.json` 的例子和一个适合在浏览器运行整个例子的 HTML 文件。

还有另外一种方式来创建 Browserify bundle：作为一个 Node 程序模块来使用 Browserify。要使用的话，需要创建一个 Browserify 实例^❶，然后告诉它你想构建什么文件^❷。

```
var browserify = require('browserify');  
var b = browserify();  
b.add('./index.js');  
b.bundle().pipe(process.stdout);
```

❶

❷

❶ 创建 Browserify 实例。

❷ 指定要构建的文件。

可以将其作为一个更复杂的构建过程的一部分来使用，或者放在一个 Grunt 任务自动化构建过程中。现在，你已经了解了如何在浏览器使用 Node 模块和如何在 Node 中模拟浏览器，接下来将学习如何优化服务器端的 web 应用程序。

9.2 服务端技术

本节内容包括用于构建 web 应用需要的技术。如果你已经在使用 Express，可以利用这些技巧来更好地组织 express 程序代码。Express 旨在保持简单和灵活，但有时候却不容易以最好的方式使用它。我们使用 Express 创建电子商务和开源的 web 应用多年所积累下来的模式和解决方案，希望能够帮助你来编写更好的 web 应用程序。

Express 3 和 4

这一章的技术内容以 Express 3 为基础，大部分代码可以在 4 中使用，或需要小部分修改。关于升级到 Express 4 的相关内容，见技巧 75。

技巧 67 Express 路由分离

Express 的文档和流行的教程通常都是把所有的代码组织在同一个文件中。在实际的项目中，这样最终会变得难以管理。本技巧会使用 Node 的模块系统把相关的路由分离成多个文件，还包括如何在不同文件中获取 Express app 对象的多种方式。

问题

主要的 Express 应用文件将变得很大，需要更好的方式来组织所有的这些路由。

解决方案

使用路由分离来把相关的路由拆分到各个模块中去。

讨论

Express 是一个简约的框架，它不会手把手地帮助你来组织项目。如果不注意，一开始很简单的项目可以变得很笨拙。成功组织好更大的项目的秘密是拥抱 Node 的模块系统。

首当其冲，要处理的便是路由，但是你可以将这个技术应用于 Express 开发的方方面面。甚至可以把应用当成是一个独立的 Node 模块，然后在其他应用将其加载进来。

这里有一个 Express 路由的普通例子：

```
app.get('/notes', function(req, res, next) {  
  db.notes.findAll(function(err, notes) {  
    if (err) return next(err);  
    res.send(notes);  
  });  
});
```

❶

```
app.post('/notes', function(req, res, next) {  
  db.notes.create(req.body.note, function(err, note) {  
    if (err) return next(err);  
    res.send(note);  
  });  
});
```

❷

❶ 这个路由会展示一个笔记列表。

❷ 这个路由是用于创建笔记的。

完整的示例项目可以在 `listings/web/route_separation/app_monolithic.js` 找到。它包括用于创建、查找和更新笔记的一组 CRUD 路由。类似这样的应用也会有其他的 CRUD 路由：也许笔记会以笔记本的形式组织起来，而且肯定有用户账户管理功能和类似设置提醒的额外功能。一旦你有了四或五组路由，这个应用文件可能就是几百行代码了。

如果你在一个单独很大的文件中编写这个项目，那么这将是很多问题的根源。因为很容易无意中使用了全局而非局部的变量导致错误，在一定条件下很危险的边缘影响也有可能遇到。Node 有一个内置的解决方案，可以应用于 Express 或者其他的 web 框架：目录即模块。

使用模块来重构路由，首先创建一个名为 `routes` 的目录，或者更倾向 `controllers` 也可以。然后创建一个 `index.js` 文件。在例子中，它是一个只有三行简单代码的文件，来把

处理笔记的路由输出给外部。

```
module.exports = {  
  notes: require('./notes')  
};
```

❶ 类似这样来输出每一个路由。

在这里，我们只有一个路由模块，使用 `require` 和相对路径来进行加载❶。接着，复制整个路由组的内容粘贴到 `routes/notes.js` 文件里。然后删除路由定义的部分——例如，`app.get('/notes', , 替换成 module.exports.index = function(req, res) { 来输出给外部。重构后的文件应该如下边代码所示：`

例子 9.6 一个不包括应用其他部分的路由模块

```
var db = require('../db');  
  
module.exports.index = function(req, res, next) {  
  db.notes.findAll(function(err, notes) {  
    if (err) return next(err);  
    res.send(notes);  
  });  
};  
  
module.exports.create = function(req, res, next) {  
  db.notes.create(req.body.note, function(err, note) {  
    if (err) return next(err);  
    res.send(note);  
  });  
};  
  
module.exports.update = function(req, res, next) {  
  db.notes.update(req.param('id'), req.body.note, function(err, note) {  
    if (err) return next(err);  
    res.send(note);  
  });  
};  
  
module.exports.show = function(req, res, next) {  
  db.notes.find(req.param('id'), function(err, note) {  
    if (err) return next(err);  
    res.send(note);  
  });  
};
```


❶ 对外每一个路由方法的名称对应了 CURD 操作。

每一个路由方法对外以一个 CRUD 风格的名称输出（index, create, update, show）❶。相应的 app.js 文件现在可以清理一下。下边的代码展示了处理后有多么简洁。

例子 9.7 重构后的 app.js 文件

```
var express = require('express');  
var app = express();  
var routes = require('./routes');
```

❶

```
app.use(express.bodyParser());
```

```
app.get('/notes', routes.notes.index);  
app.post('/notes', routes.notes.create);  
app.patch('/notes/:id', routes.notes.update);  
app.get('/notes/:id', routes.notes.show);
```

❷

```
module.exports = app;
```

❸

❶ 加载多个路由。

❷ 将路由和 HTTP 动词及部分 URL 进行绑定。

❸ 对外输出 app 对象。

所有路由会在 require('./routes') 时立刻加载❶。这样简单方便，因为会减少 require 语句，否则 app.js 会变得凌乱。你所需要做的便是移除旧的路由 callback，把每一个路由的方法引用添加进去❷。

不要在这个文件里放一个 app.listen 的方法，而是对外暴露 app❸。这样会让测试应用更加容易。对外暴露 app 对象另外一个优点是你可以应用的其他任意地方加载 app.js 模块。Express 允许你获取和设置配置的值，如果想要在路由意外的其他地方引用这些值，让 app 可以访问是非常有用的。同样地，注意一下，res.app 是在路由内部访问的，所以不用经常传递 app 对象。

如果你想很容易地加载 app.js，而不用创建服务器，那么把应用文件命名为 app.js，然后分出来一个 server.js 文件来调用 app.listen。你可以在 package.json 中设置 server 属性来使用 node server.js，这样便可以让别人使用 npm start 来启动应用——你也可以不配置 server 属性，因为 node server.js 是默认值，但最好还是定义一下以便别人知道你希望他们去使用它。

目录即模块

这一节的技巧把所有的路由放在一个目录下，然后使用一个 `index.js` 文件来输出，以便它们可以在另外一个地方使用 `require('./routes')` 来加载。

这个模式也可以在其他地方使用。它可以很好地组织中间件、数据库模块和配置文件。

使用目录即模块的方式来组织配置文件的例子，可以参考技巧 69。

这个技术的完整例子可以在 `listings/web/route-separation` 找到，它包括了一些测试的例子，如果你需要在自己的项目中做单元测试可以参考。

合理地组织 Express 项目是很重要的，但是还有其他工作流相关的问题会导致开发缓慢。例如，当你在开发 web 应用时，通常会做一些小修改然后刷新浏览器来查看结果。大部分 Node 框架要求进程重启后才能看到修改生效，所以下一个技巧会探索如何有效地解决这个问题来更好地进行工作。

技巧 68 自动重启服务器

虽然 Node 已经自带了监测文件改动的工具，但还是需要大量的工作来让它们可以有效工作。这个技巧着眼于 `fs.watch`，并介绍了常用的第三方工具，来让文件编辑保存时可以自动重启 web 应用。

问题

你需要在每次修改文件时重新启动 Node 应用程序。

解决方案

使用文件监听器来自动重启应用。

讨论

如果你已经习惯了像 PHP 或 ASP 的语言，那么 Node 的进程内置基础服务器的模型看起来有点非同寻常。Node 模型其中最大的不同点是当文件修改时需要重启进程。如果你理解 `require` 和 V8 引擎是如何工作的，就明白这是有意义的——文件通常是加载和解析一次。

解决这个问题的方法之一是监测文件修改，然后重启应用。Node 充分利用了非阻塞的 I/O，非阻塞文件系统的 APIs 的一个属性是事件监听器可以用于等待特定的事件。为了

解决这个问题，你可以为项目中的所有文件来设置文件系统的事件处理器。那么，当文件修改时，你的事件处理器便会重启这个项目。

Node 在 `fs` 模块中提供一个 API `fs.watch` 来处理这个。在编写本书时，这个 API 是不稳定的——这意味着它可能在随后的 Node 版本会修改。这个方法在第 6 章 6.1.4 节的内容有涉及到。我们看一下它如何在一个 web 应用中使用。例子 9.8 展示了一个程序可以用于监测和重启一个简单的 web 服务器。

例子 9.8 重新加载一个 Node 进程

```
var fs = require('fs');
var exec = require('child_process').exec;

function watch() {
  var child = exec('node server.js');
  var watcher = fs.watch(__dirname + '/server.js', function(event) {
    console.log('File changed, reloading.');
```

❶

```
    child.kill();
    watcher.close();
    watch();
  });
}

watch();
```

❷
❸
❹
❺

- ❶ 开启 web 服务器进程。
- ❷ 使用 `fs.watch` 来监测文件修改。
- ❸ 当文件修改时，关闭 web 服务器。
- ❹ 关闭监测器。
- ❺ 递归调用监测器函数来再次开启服务。

使用 `fs.watch` 来监测一个文件的修改是稍微有些难理解，但是你可以使用 `fs.watchFile`，这是基于文件轮询而非 I/O 事件。例子 9.8 的方法是，开启一个子进程——这个例子是 `node server.js` ❶——然后监听文件的修改 ❷。开始和停止进程是使用 `child_process` 核心模块来管理，`kill` 方法可以用于停止子进程 ❸。

在 Mac OS 中最好也使用 `watcher.close` 来停止文件的监测 ❹，尽管 Node 的文章说明了 `fs.watch` 应该是“持续的”。一旦这一切都处理好了，`watch` 函数会递归调用来再次加载 web 服务器 ❺。

这个例子可以用这样的 `server.js` 运行：

```
var http = require('http');
var server = http.createServer(function(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('This is a super basic web application');
});
```

```
server.listen(8080);
```

这个做法，不是特别优雅，同时也并不完整。大部分 Node web 应用由多个文件组成，所以文件监测逻辑会变得比较复杂。这并不能直接递归所有的父级目录，因为有很多文件你是不想监测的——你不想监测 `.git` 目录下的文件，或者编写一个 Express 应用你可能不想监测视图层的模板文件，因为它们在开发者模式下会强制加载而不使用缓存。

突然自动重启 Node 程序不是很好，但有其他第三方模块可以帮忙处理。其中解决这个问题方案中使用最广泛的一个是 Remy Sharp 的 `nodemon` (<http://nodemon.io>)。它运作良好，对监测 Express 程序来说开箱即用。你甚至可以使用它来重启任何程序，无论是 Node、Python 或 Ruby 等。

来尝试使用它，输入 `npm install -g nodemon`，然后到包含了 Node web 应用的目录中。如果你想要使用一个小的脚本例子，可以使用我们的例子，`listings/web/watch/server.js`。

输入 `nodemon server.js` 来开始运行和监测 `server.js`，你会发现在 `res.end` 编辑文本后下一次加载 `http://localhost:8080/` 便可以看到改动。

你可能会注意到在改动之前有一个小延迟——那是 `Nodemon` 设置 `fs.watch` 或 `fs.watchFile`，如果它在你的操作系统中不可用。你可以输入 `rs` 然后回车来强制重新加载。

`Nodemon` 有很多其他的特性来帮你更好地开发 web 应用。输入 `nodemon --help` 可以看到命令选项的列表，但是你可以通过创建一个 `nodemon.json` 文件来获得更好的，对 VCS 友好的控制。这允许你指定一个忽略的文件数组，并且也可以使用 `execMap` 把文件扩展映射到程序名称上。`Nodemon` 的文档包括了一个例子文件来说明每一个特性。

接下来是一个 `Nodemon` 配置的例子，你可以在自己的应用中使用它。

例子 9.9 Nodemon 的配置文件

```
{
  "ignore": [
    ".git",
    "node_modules/node_modules"
  ],
```

```
"execMap": {  
  "js": "node --harmony"  
},  
"watch": [  
  "test/fixtures/",  
  "test/samples/"  
],  
"env": {  
  "NODE_ENV": "development"  
},  
"ext": "js json"  
}
```

- ❶ 忽略的路径列表。
- ❷ 自动映射.js 文件来让运行 Node 时使用 harmony 标识。
- ❸ 指定检测的路径。
- ❹ 环境变量列表。

基本的选项允许你忽略指定的路径❶，并且列出多个要检测的路径❸。这个例子使用 execMap 来为 node 执行所有 JavaScript 文件❷时使用 --harmony 标识¹。Nodemon 也可以设置环境变量——只需要在 env 属性中添加一些值❹。

一旦你的工作流程得到简化，应该感谢 Nodemon，接下来的事情就是提高项目的可配置性。大多数项目需要多个层级的配置——例如数据库连接的细节和远程 APIs 的授权证书。接下来的技巧着眼于如何配置你的 web 应用，可以轻松地将其部署到多个环境，或者在测试环境中运行，甚至是在本地开发时调整它的行为。

技巧 69 配置 web 应用

这种技术着眼于共同的模式来配置 Node 的 web 应用程序。我们将通过列举 Express 实例，但你可以在其他 Web 框架使用这些模式。

问题

你需要在开发环境、测试环境、生产环境有不同的配置。

解决方案

使用 JSON 配置文件、环境变量或者一个模块来进行管理。

¹ --harmony 用于开启 Node 中可用的所有新的 ECMAScript 特性。

讨论

大多数 web 应用程序需要一些配置值，来确保这些操作是正确的：如数据库中连接字符串、缓存设置和电子邮件服务器证书等，都是典型的例子。有许多方法来存储应用程序设置，但在你安装第三方模块之前，考虑你的需求：

- 在版本控制中存放数据库信息是否可以接受？
- 是否真的需要配置文件，或者可以把配置嵌入应用程序中？
- 如何在应用程序的不同部分中访问配置值？
- 你的部署环境是否提供了一种存储配置的方式？

第一点要看你项目的组织和策略。如果正在构建一个开源的 web 应用，你不想把数据库账号信息存放在公共代码仓库中，那么配置文件可能不是最好的解决方案。你希望人们快速简单地安装应用程序，但是你不希望泄漏密码。类似地，如果你在一个大的组织工作，有着数据库管理员，他们可能不会轻易让每个人都可以直接访问数据库。

在这些例子中，你可以把配置值作为部署环境的一部分。环境变量是一种标准的方式来配置 UNIX 和 Windows 程序的行为，你可以使用 `process.env` 来获取它们。最基础的例子是使用 `NODE_ENV` 设置在多个部署环境中切换。下面列出了在 Express 中使用配置的模式。

例子 9.10 配置 Express 应用

```
var express = require('express');
var app = express();

app.set('port', process.env.PORT || 3000);

app.configure('development', function() {
  app.set('db', 'localhost/development');
});

app.configure('production', function() {
  app.set('db', 'db.example.com/production');
});

app.listen(app.get('port'), function() {
  console.log('Using database:', app.get('db'));
  console.log('Listening on port:', app.get('port'));
});
```

❶ 如果环境变量 `PORT` 未设置时使用 3000 端口。

❷ 这个回调函数仅在 `NODE_ENV` 设置为 `development` 时才会执行。

❸ 这个回调函数仅在 `NODE_ENV` 设置为 `production` 时才会执行。

❹ 使用 `app.get()` 来获取配置。

Express 有一个简单的 API 用于设置应用程序的配置：`app.set`、`app.get` ❹ 和 `app.configure`。你还可以使用 `app.enable` 和 `app.disable` 来进行切换布尔类型的值，而 `app.enabled` 和 `app.disabled` 用来进行查询。`app.configure` 相当于 `if (process.env.NODE_ENV === "development")` ❷ 和 `if (process.env.NODE_ENV === "production")` ❸，所以如果不想使用 `app.configure` 时没必要使用。在 Express 4 中可以移除。如果没有使用 Express，你只需要查询 `process.env` 就可以了。

`NODE_ENV` 环境变量由 shell 控制，如果你想让例子 9.10 的代码在生产环境模式下运行，可以输入 `NODE_ENV=production node config.js`，你应该可以看到它打印出生产环境数据库的字符串。你也可以输入 `export NODE_ENV=production`，在当前 shell 运行的情况下，这可以让应用程序都以生产模式运行。

我们使用 `PORT` ❶ 来设置端口是因为这是 Heroku 默认使用的名称。这允许 Heroku 的内部 HTTP 路由来处理你应用监听的端口。

你可以在所有代码中使用 `process.env` 来替代 `app.get`，但是使用 `app` 对象会更加简洁一些。你不需要到处传递 `app`——如果你已经使用技巧 67 的路由分离模式，那么可以通过 `res.app` 来访问它。

如果你想使用配置文件，最简单、最快捷的方式是使用目录和 JSON 文件来作为模块。创建一个名为 `config/` 的目录，然后创建一个 `index.js` 文件，和为对应每一个环境创建一个 JSON 文件。下边例子展示了 `index.js` 文件应该是怎么样的：

例子 9.11 一个 JSON 配置文件头部

```
var config = {  
  development: require('./development.json'),  
  production: require('./production.json'),  
  test: require('./test.json')  
};  
  
module.exports = config[process.env.NODE_ENV || 'development'];
```

❶ 使用 `require()` 来加载 JSON 文件。

❷ 检查 `NODE_ENV` 来确定使用哪一个文件。

Node 的模块系统允许你使用 `require` 来加载 JSON 文件^❶，所以你可以加载每个环境的配置文件，然后对外暴露出和 `NODE_ENV` 相关的配置^❷。无论何时需要访问配置，只需使用 `var config = require('./config')`——你会得到一个普通的 JavaScript 对象包含了当前环境的配置。下边例子展示了 Express 使用该技巧的例子。

例子 9.12 加载配置的目录

```
var express = require('express');
var app = express();
var config = require('./config');❶

app.listen(config.port, function() {
  console.log('Using database:', config.db);
  console.log('Listening on port:', config.port);
});
```

❶ 使用 `require()` 加载配置。

这是如此简单以至于看起来就像假的一样。所有你需要做的就是调用 `require('./config')`，然后便可以拿到配置。Node 的模块系统会把文件缓存起来，每一次执行 `require`，并没有再次重新加载 JSON 文件。你可以在应用的任意地方反复调用 `require('./config')`。

这个技术利用了 JavaScript 设置和访问对象值的轻量级语法，以及 Node 的模块系统。它适用于很多类型的项目。

还有另外一个管理配置的方法：使用第三方模块。对于这最后一点，你可能感觉有点过了，但是第三方模块可以提供很多功能，包括命令行选项解析。你可能需要经常在不同选项之间切换，因此使用命令行选项来修改应用配置也会让你感兴趣。

web 框架 Flation (<http://flatironjs.org/>) 有一个名为 `nconf` 的可应用的配置化模块 (<https://npmjs.org/package/nconf>) 来处理配置、环境变量，以及命令行选项。可以优先考虑命令行，这样就可以使命令行选项覆盖配置文件。它是一个处理选项的统一框架。

下面显示 `nconf` 如何被用于配置 Express 应用程序。

例子 9.13 使用 `nconf` 来配置 Express 应用

```
var express = require('express');
var app = express();
var nconf = require('nconf');
var routes = require('./routes');
```

```

nconf
  .argv()
  .env()
  .file({ file: 'config.json' });

nconf.set('db', 'localhost/development');
nconf.set('port', 3000);

app.get('/', routes.index);

app.listen(nconf.get('port'), function() {
  console.log('Using database:', nconf.get('db'));
  console.log('Listening on port:', nconf.get('port'));
});

```

❶ 让 nconf 选择性地使用配置文件，命令行优先级最高，可以覆盖其他。

❷ 设置默认的数据库配置。

❸ 获取端口号。

在这里我们可以告知 nconf 优先使用命令行选项，如果有配置文件可用时便读取配置文件❶。不需要创建一个配置文件，使用 nconf.save 时，nconf 会为你创建一个。这意味着你可以在应用程序中更改配置，然后保存。当 nconf 被设置为使用数据库来保存配置时是最棒的——它也内置了 Redis 的支持。

默认的值可以使用 nconf.set 来设置❷。如果你不带任何配置选项来允许这个例子，那么它会使用 3000 端口，但如果使用 node app.js --port 3001 来启动它，那么它会使用你传递的 --port 选项。获取配置很简单，使用 nconf.get 即可❸。

同时，你不需要到处传递 nconf 对象。配置在内存中保存着。在你项目的任何文件可以使用 require 加载 nconf，然后调用 nconf.get 来访问配置。下面是再次加载 nconf，然后尝试访问 db 配置的代码。

例子 9.14 在应用的任意地方加载 nconf

```

var nconf = require('nconf');

module.exports.index = function(req, res) {
  res.send('Using database:', nconf.get('db'));
};

```

❶ 如果再次加载 nconf，它会知道要做什么。

虽然 var nconf = require('nconf') 看起来好像会返回 nconf 的一个副本，但是它没有❶。

一个组织良好的、精心配置的 Web 应用程序仍然可能出错。当你的应用程序崩溃，你会需要日志来帮助调试问题。接下来的技术将帮助你更好地把握应用程序如何处理错误。

技巧 70 优雅地处理错误

这一节内容主要是在应用中如何使用 Error 构造函数来捕获和处理错误。

问题

你想要统一处理错误异常来简化 web 应用程序。

解决方案

从 Error 继承包括 HTTP 状态码的错误类，并使用中间组件来处理基于 content type 的错误。

讨论

JavaScript 有 Error 的构造函数，你可以继承它来表示特定类型的错误。在 web 开发中，一些错误越来越频繁地出现了：不正确的网址、不正确的参数查询参数或表单值、并发错误。这意味着你可以定义一些错误，来包括典型错误所提供的 HTTP 代码。

你应该调用 next(err)，而不是在 HTTP 路由中分离出一个错误条件的处理分支。接下来的代码会展示这一切是如何工作的。

例子 9.15 传递错误给中间件

```
var db = require('../db');  
var errors = require('../errors');  
  
module.exports.show = function(req, res, next) {  
  db.notes.find(req.param('id'), function(err, note) {  
    if (err) return next(err);  
    if (!note) {  
      return next(new errors.NotFound('That note was not found.'));  
    }  
    res.send(note);  
  });  
};
```

- ① 在分离的文件中组织错误对象。
- ② 确保路由的处理声明中包括了第三个参数，next。
- ③ 如果数据库 API 传递了一个错误，那么便提交返回结果。

❷ 如果找不到记录，创建一个合适的错误实例。

在这个例子中，处理错误的类已经在单独的文件中定义❶，你可以在例子 9.16 中找到。这个路由处理包括了第三个参数，next ❷，在我们之前提到标准的 req、res 之后。

许多路由的处理程序会加载数据库中的数据，无论是从 MySQL、PostgreSQL、MongoDB 或者 Redis，所以这个例子是基于一个通用的数据库异步 API 的。如果数据库 API 抛出一个错误，那么提早返回，并且将错误对象作为第一个参数调用 next。这会把错误传递给下一个中间件❸。这个路由处理器有另外一部分的逻辑——如果在数据库中找到记录，那么会实例化一个错误对象并使用 next 进行传递❹。

下边的列表展示了如何继承一个 Error。

例子 9.16 继承错误类来包含错误码

```
var util = require('util');

function HTTPError() {
  Error.call(this, arguments);
}

util.inherits(HTTPError, Error);

function NotFound(message) {
  HTTPError.call(this);
  Error.captureStackTrace(this, arguments.callee);
  this.statusCode = 404;
  this.message = message;
  this.name = 'NotFound';
}

util.inherits(NotFound, HTTPError);

module.exports = {
  HTTPError: HTTPError,
  NotFound: NotFound
};
```

❶ 创建通用的 HTTP 错误类。

❷ 使用 util.inherits 从 Error 继承。

❸ 捕获堆栈信息。

❹ 设置可以传递给浏览器的状态码。

❺ 新增的 HTTP 错误可以从 HTTPError 继承而来。

这里我们有选择性地创建了两个类。不仅仅只是定义了 `NotFound` 错误，还创建了 `HTTPError` 类^❶，并且从它继承而来^❷。这方便于我们跟踪错误是否与 HTTP 相关，或者它也可能是其他错误。`HTTPError` 这个基础的错误类可以从 `Error` 继承而来^❸。

在 `NotFound` 错误中，我们捕获了堆栈跟踪信息，来帮助调试^❹，同时设置了一个可以反馈给浏览器的状态码^❺。

下面显示了如何在一个典型的 Express 应用程序中创建一个错误处理的中间件。

例子 9.17 使用一个处理错误的中间件

```
var errors = require('./errors');
var express = require('express');
var app = express();
var routes = require('./routes');

app.use(express.bodyParser());

app.get('/notes/:id', routes.notes.show);

app.use(function(err, req, res, next) {❶
  if (process.env.NODE_ENV !== 'test') {❷
    console.error(err.stack);
  }

  res.status(err.statusCode || 500);

  res.format({
    text: function() {
      res.send(err.message);❸
    },

    json: function() {
      res.send(err);
    },

    html: function() {
      res.render('errors', { err: err });
    }
  });
});

module.exports = app;
```

❶ 如果在 `app.use` 使用了四个参数，那么第一个参数是错误对象。

❷ 如果不是在测试模式中便打印堆栈跟踪信息。

❸ 以期望的格式返回错误信息。

这个中间件相对简单，但是它已经包括了我们在生产环境中发现的一些需要调整的东西。为了在 next 中获取错误对象，请确保使用 app.use 回调函数的四个参数❶。同时请注意这个中间件在处理链条的最后，需要把它放在其他中间件和路由的定义之后。

可以选择性地打印堆栈跟踪信息，当特定的测试需要依赖错误信息时❷——错误作为测试的一部分被触发，堆栈信息应该是不可见的，你不希望堆栈信息搞乱了测试的输出。

因为需要在主要的应用文件中集中处理错误，所以根据条件返回不同的错误格式是一个很好的主意。如果应用提供 HTML 页面以及 JSON API，那么这会是非常有用的。你可以使用 app.format 来处理❸，它用于检查在请求的 Accept 头部中的 MIME type。JSON 响应可能并不需要，但是你的 API 需要返回客户端可以预知的符合规则的错误信息——如果当你请求 JSON 时突然响应的是 HTML，那么会很难处理。

在测试中，应该检查这些错误是否是你所需要的。下边的代码片段展示了 mocha 测试来确保 404 时以所期望的格式返回了所期望的内容：

```
describe('Error handling', function() {
  it('should return a 404 for IDs that do not exist', function(done) {
    request(app)
      .get('/notes/999')
      .expect(404, done);
  });

  it('should send JSON errors when requested', function(done) {
    request(app)
      .get('/notes/999')
      .set('Accept', 'application/json')
      .expect(404, function(err, res) {
        assert.equal(res.body.name, 'NotFound');
        done();
      });
  });
});
```

❶

❷

❸

❶ 检查返回的状态码是否为期望的值。

❷ 设置 Accept 头部来获取 JSON。

❸ 检查 body 是否是期望的格式。

这个代码片段包括了两个请求。第一个用例是我们得到了一个 404 错误❶，第二个是设置 Accept 头部来确保我们获取的是 JSON❷。这是 SuperTest 的一个实现，将在响应中返回 JSON，断言可以检查来确保我们拿到的对象是我们期望的格式❸。这个例子的源码可以在 listings/web/error-handling 中找到。

错误的电子邮件表格

如果你让你的应用程序在异常情况发生时发送邮件通知，这里有一个表，列出了你应该在邮件中包含的信息，来帮助调试：

- 错误对象的字符串版本。
- `err.stack` 的内容——这是 Node 在错误对象中的非标准属性。
- 请求方法和 URL。
- Express `req.route` 属性，如果有的话。
- 远程的 IP，在 Express 是 `req.ip`。
- 请求的 body，你可以使用 `inspect(req.body)` 来转换为字符串。

这种错误处理的模式在 Express 应用中被广泛使用，甚至它已经在 `restify` (<https://npmjs.org/package/restify>) 中内置了。如果你记得把错误对象传递给 `next`，那么可以更早地测试和调试 Express 应用程序。

错误也可以使用传输脚本以 Email 发送。为了最大化异常错误邮件的作用，你需要在邮件中包含请求和错误对象以便你分析是哪里出了问题。同时，你也可能不希望发送关于某些状态码的错误细节，这取决于你。

在这个技术中我们提到了让代码适应使用 REST API 工作。下一个技巧会参考 Express 和 `restify` 的一些例子，来深入到 REST 的世界。

技巧 71 RESTful web 应用

某些情况下你可能需要添加一个 API 到应用程序中去。这一节讲述的技术都是关于构建 RESTful API 的。这里提供了 Express 和 `restify` 的例子，以及一些提示，来帮助你如何使用正确的 http 动词和同构化的 URLs 来创建 APIs。

问题

你想要使用 Express、`restify` 或者其他框架来创建 RESTful 的 web 服务。

解决方案

使用正确的 HTTP 方法、URLs 和头部信息来创建语义化的 RESTful API。

讨论

REST 即为表征性状态传输，²记住这个并不是特别有用，除非你想在面试中打动别人。web 开发人员讨论它时通常会和 SOAP（简单对象访问协议）做对比，通常被视为是更有协作性和更严谨的一种创建 web APIs 的方式。事实上，REST API 是相对严格的，但是关键的区别在于 REST 在基础的层面上拥抱 HTTP——HTTP 方法本身就具备了语义。

如果你曾经做过基础表单，应该对 GET 和 POST 请求很熟悉。在 REST 中，这些 HTTP 动词有着特定的含义。例如，POST 用于创建资源，GET 表示获取资源。

Node 开发者通常使用 JSON 来创建 API。在 Node 中，JSON 是最便于生成和读取的结构化数据格式，它同时在客户端的 JavaScript 也能很好地工作。但是 REST 不意味着 JSON——你可以自由地使用任意数据格式。一些客户端和服务使用 XML，我们甚至看到过使用 CSV 和电子表格格式，如 Excel。

所期待的数据格式可以在请求的 Accept 头部中指定。对于 JSON，它应该是 application/json，而 XML，则是 application/xml。还有其他有用的请求头部——Accept-Version 可以用于请求不同版本的 API。这允许客户端来使用自己支持的版本，同时你可以随意地升级服务器而不破坏向后兼容性——你总是可以在人们更新客户端之前更快地更新服务器程序。

Express 基于 http 核心模块提供了一个轻量的封装层，但它不包括任何数据持久化，除了在内存中的 sessions 和 cookies。你必须决定使用哪个数据库和数据库模块。restify 也是如此：它不会自动从 HTTP 中映射数据来做离线存储，你需要自己找个方式来处理。

Restify 和 Express 十分相似。其中不同的是 Express 有构建 web 应用的特性，包括了渲染模块。相反地，restify 致力于构建 REST APIs，这带来了不同的需求。Restify 可以让使用语义化的 HTTP 版本头部信息来处理多个版本的 APIs 变得容易，并且拥有基于事件的 API 来触发和监听与 HTTP 相关的事件及错误。它也支持节流，你可以控制多快来处理一次响应。

图 9.3 中展示了一个典型的 RESTful API，允许 page 对象创建、读取、更新和删除。

要开始构建 REST APIs，应该考虑你的对象是怎么样。想象一下你在构建一个内容管理系统：可能有页面、用户和图片。如果要添加一个按钮来允许页面在“发布”和“草

²更多关于 REST 的内容，参考 Fielding 相关的论文：<http://mng.bz/7Fhj>。

稿”之间切换，可能需要一个 REST API，它支持这样的请求：PATCH /pages/:id，你可以把按钮和一些客户端的 JavaScript 绑定，或者是一个表单携带 { state: 'published' } 或 { state: 'draft' } 数据 post 提交到 /pages/:id。如果你面临的是一个 Express 应用程序，只有 PUT /pages/:id，那么可能需要把现有的实现改为使用 PATCH 的代码。

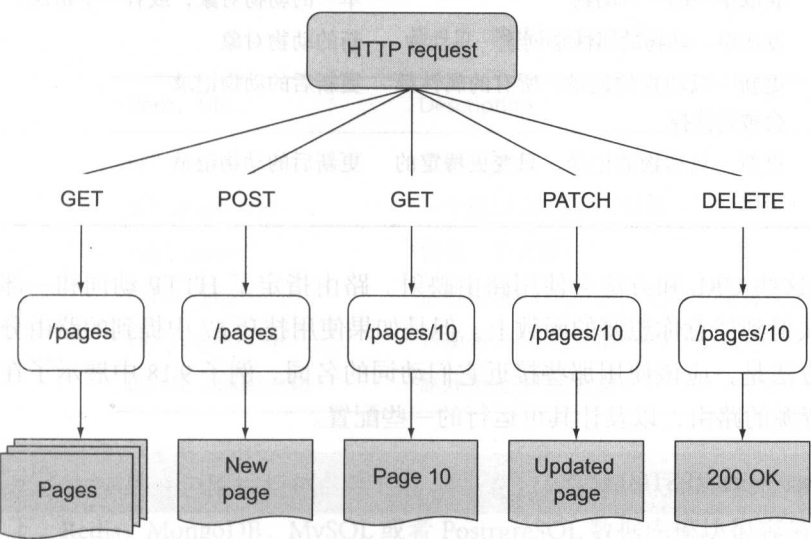


图 9.3 创建 REST API 的请求

复数还是单数？

在设计 API 的 URI 路径 (endpoints) 时，你应该使用复数的名词。这意味着 /pages 和 /pages/1 都是一个指定的页面，而非 page/1。如果路径都保持一致，使用你的 API 也会更加简单一些。

你可能会发现有一些资源，应该是单数名词，因为只有这样一个东西。如果要更加符合语义化，使用单数名词，否则统一使用复数。例如，如果你的 API 要求用户登录，并且不希望暴露唯一的用户 ID，那么如果仅有一个给定的用户账号 /account 对于用户账号管理来说，是明智的终点。

表 9.1 展示了 HTTP 动词和对应的典型响应。需要注意的是 PUT 和 PATCH 有着不同但很相似的含义——PATCH 意味着修改资源的某一部分，而 PUT 则是替换整个资源。以这种方式来构建应用需要一些练习，但是它更加切合实际情况且容易测试，是值得好好学习的。如果这些 HTTP 内容对你来说是新的，那么在设计应用的 API 时可以参考表 9.1。

表 9.1 选择正确的 HTTP 动词

Verb	Description	Response
GET /animals	获取动物列表	动物对象的数组
GET /animals/:id	获取单一的一只动物	单一的动物对象，或者一个错误
POST /animals	发送单一动物的属性来创建一只动物	新的动物对象
PUT /animals/:id	更新一只动物的记录，所有的属性都会被替换掉	更新后的动物记录
PATCH /animals/:id/	更新一只动物的记录，只变更特定的字段	更新后的动物记录

在 Express 应用中，这些 URL 和方法会使用路由映射。路由指定了 HTTP 动词和一部分的 URL。你可以映射到任意你想要的函数上，但是如果使用技巧 67 中提到的路由分离模式，更可取的方法是，应该使用那些接近它们动词的名词。例子 9.18 中展示了在 Express 中 RESTful 资源的路由，以及让其可运行的一些配置。

例子 9.18 在 Express 中的 RESTful 资源

```
var app;
var express = require('express');
var routes = require('./routes');

module.exports = app = express();

app.use(express.json());
app.use(express.methodOverride());

app.get('/pages', routes.pages.index);
app.get('/pages/:id', routes.pages.show);
app.post('/pages', routes.pages.create);
app.patch('/pages/:id', routes.pages.patch);
app.put('/pages/:id', routes.pages.update);
app.del('/pages/:id', routes.pages.remove);
```

- ❶ 使用 JSON body 解析。
- ❷ methodOverride 中间件允许一个查询参数来指定额外的 HTTP 方法。
- ❸ 资源使用的路由。

这个例子使用了一些中间件来自动解析 JSON 请求❶和用 _method 的查询参数来覆盖 HTTP 的方法 POST ❷。这意味着 PUT、PATCH、DELETE 这些 HTTP 动词可以使用 _method

查询参数来确定。这是因为大多数浏览器只能发送 GET 或者 POST 请求，所以 `_method` 是一个大多数 web 框架使用的替代方法。

例子 9.18 中的路由定义了通常 RESTful 资源用到的每一个方法❸。表 9.2 展示了这些路由如何映射到动作上。

表 9.2 响应的路由映射

Verb, URL	Description
GET /pages	页面的数组
GET /pages/:id	一个指定 id 的页面对象
POST /pages	创建一个页面
PATCH /pages/:id	加载指定 id 的页面，更改一些属性字段
PUT /pages/:id	替换指定 id 的页面
DELETE /pages/:id	删除指定 id 的页面

例子 9.19 是一个路由处理的实现实例。它使用了基于 Node 数据库的 API——真实情况下，Redis、MongoDB、MySQL 或者 PostgreSQL 数据库模块很容易就可以获取到，并且很容易上手。

例子 9.19 RESTful 的路由处理器

```
var db = require('../db');

module.exports.index = function(req, res, next) {
  db.pages.findAll(function(err, pages) {
    if (err) return next(err);
    res.send(pages);
  });
};

module.exports.create = function(req, res, next) {
  var page = req.body.page;
  db.pages.create(page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.update = function(req, res, next) {
```

```
var id = req.param('id');
var page = req.body.page;
db.pages.update(id, page, function(err, page) {
  if (err) return next(err);
  res.send(page);
});
};

module.exports.show = function(req, res, next) {
  db.pages.find(req.param('id'), function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.patch = function(req, res, next) {
  var id = req.param('id');
  var page = req.body.page;
  db.pages.patch(id, page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.remove = function(req, res, next) {
  var id = req.param('id');
  db.pages.remove(id, function(err) {
    if (err) return next(err);
    res.send(200);
  });
};
```

❶ 如果数据库抛出错误，那么跳到下一个中间件。

❷ 调用 `send` 来自动返回 JSON 给浏览器。

❸ 大部分数据库模块不会有名为 `patch` 的方法，但有类似的方法。

虽然这个例子很简单，但是它说明了一些很重要的东西：你应该让路由处理保持轻量。它们处理 HTTP 请求，然后让一些其他部分的代码来处理业务逻辑。另外在例子中用到的模式是错误处理——错误会调用 `next(err)` 来进行传递❶。尝试让错误处理的代码保持集中和通用——技巧 70 中有更多相关的详细内容。

传入一个 JavaScript 对象给 `res.send()` 来返回 JSON 给浏览器❷。Express 知道如何把对象转换为 JSON，所以这就是你需要做的一切。

所有的这些路由处理器都使用了同一个模式：把查询参数或者请求 body 映射到数据库可以使用的某些东西上，然后调用对应的数据库方法。如果你正在使用 ORM 或者 ODM——一种更加抽象的数据库层——那么你很可能已经有一些类似 PATCH 的方法^③。这可以是一个 API 方法，允许你只更新指定的区域内容。关系型数据库和 MongoDB 都有这种方法。

若下载了本书的源代码，可以拿到尝试运行完整例子的其他文件。输入 `npm start` 来运行。当服务器运行后，你可以使用下边的 `Curl` 命令来和服务器进行交互。

第一个命令用于创建一个页面：

```
curl -H "Content-Type: application/json" \  
  -X POST -d '{ "page": { "title": "Home" } }' \  
  http://localhost:3000/pages
```

①
②
③

① 使用 JSON body 的编码格式。

② 方法是 POST。

③ 创建页面的 URL 是 /pages。

首先使用 `-H` 选项来指定 `Content-Type` ①。然后，设置请求使用 `POST`，请求的 body 包含了一个 JSON 字符串②。URL 是 /pages，因为我们正要创建一个资源③。

当你能够理解 `Curl` 的配置项时，它会是一个探索 API 很有用的工具。我们先记住 `-H` 是设置头部，`-X` 是设置 HTTP 方法，`-d` 是设置请求 body 的内容。

要查看页面的列表，只需要使用 `curl http://localhost:3000/pages`。要修改内容的话，尝试一下 `PATCH`：

```
curl -H "Content-Type: application/json" \  
  -X PATCH -d '{ "page": { "title": "The Moon" } }' \  
  http://localhost:3000/pages/1
```

Express 有一些其他的好东西来帮助创建 RESTful 的 web 服务。记得有一些 REST API 会使用其他格式的数据么，如 XML？如果你两者都想要怎么办？你可以通过使用 `res.format` 来解决这个问题。

```
module.exports.show = function(req, res, next) {  
  db.pages.find(req.param('id'), function(err, page) {  
    if (err) return next(err);  
    res.format({  
      json: function() {  
        res.send(page);  
      },  
    },
```

①
②

```

    xml: function() {
      res.send('<page><title>' + page.title + '</title></page>');
    }
  });
});
};

```

- ❶ format 方法接收一个对象。
- ❷ json 是 application/json 的缩写。
- ❸ 每一个内容类型对应了一个处理方法。

使用 XML 来替代 JSON，你在请求头部中设置对应的 Accept 头部信息即可。使用 Curl 的话，可以这么做：

```

curl -H 'Accept: application/xml' \
  http://localhost:3000/pages/1

```

记住 Accept 是用于向服务器请求特定格式用的，而 Content-Type 是告诉服务器说正在发送的内容是什么格式。有时候两个同时包含在同一个请求中是有意义的。

现在你已经看到了 REST APIs 如何在 Express 中工作，我们可以和 restify 对比一下。在 Express 应用结构中使用的模式是可以在 restify 项目中复用的。其中两个重要的模式是路由分离，在技巧 67 中有详细描述，以及应用在服务器的一个单独的文件中定义（方便测试和内部复用）。例子 9.20 是例子 9.18 的代码改为使用 resify 实现。

例子 9.20 一个 restify 应用

```

var app;
var restify = require('restify');
var routes = require('./routes');

module.exports = app = restify.createServer({
  name: 'NIP CMS',
});

app.use(restify.bodyParser());

app.get('/pages', routes.pages.index);
app.get('/pages/:id', routes.pages.show);
app.post('/pages', routes.pages.create);
app.patch('/pages/:id', routes.pages.patch);
app.put('/pages/:id', routes.pages.update);
app.del('/pages/:id', routes.pages.remove);

```

- ❶ 创建 restify 服务实例。
- ❷ 使用中间件来解析 JSON。
- ❸ 设置路由。

使用 restify，服务器的实例会使用一些初始化的配置来创建❶。你可以不传入任何选项，但这里我们指定了一个名称。这些选项通常和 Node 内置的 `http.Server.listen` 的是一样的，所以如果需要加密，可以传入 SSL/TLS 证书相关的配置。Restify 有一些选项是 Express 中不具备的，例如 `formatters`，可以让你设置使用自定义内容类型时，`res.send` 所调用的函数。

这个例子使用 `bodyParser` 来解析请求 `body` 中的 JSON❷。这和上一个例子中的 Express 中间件是一样的。

路由定义和 Express 的例子也是一样的❸。路由的回调函数会有一些小小的差异。例子 9.21 是例子 9.19 的代码使用 restify 实现。看看你是否能够认出其中的差别。

例子 9.21 Restify 路由

```
var db = require('../db');
```

```
module.exports.index = function(req, res, next) {  
  db.pages.findAll(function(err, pages) {  
    if (err) return next(err);  
    res.send(pages);  
  });  
};
```

```
module.exports.create = function(req, res, next) {  
  var page = req.body.page;  
  db.pages.create(page, function(err, page) {  
    if (err) return next(err);  
    res.send(page);  
  });  
};
```

```
module.exports.update = function(req, res, next) {  
  var id = req.params.id;  
  var page = req.body.page;  
  db.pages.update(id, page, function(err, page) {  
    if (err) return next(err);  
    res.send(page);  
  });  
};
```

```
module.exports.show = function(req, res, next) {
  db.pages.find(req.params.id, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.patch = function(req, res, next) {
  var id = req.params.id;
  var page = req.body.page;
  db.pages.patch(id, page, function(err, page) {
    if (err) return next(err);
    res.send(page);
  });
};

module.exports.remove = function(req, res, next) {
  var id = req.params.id;
  db.pages.remove(id, function(err) {
    if (err) return next(err);
    res.send(200);
  });
};
```

- ❶ 回调函数的参数和 Express 很相似。
- ❷ 获取 URL 参数会有一些差别。
- ❸ 使用 send() 直接传入整数是返回状态码。

首先要注意的一点是路由处理器的回调函数参数是和 Express 一样的❶。事实上，你几乎可以直接把 Express 应用的代码拿过来使用。但是还是有些许区别的：没有 req.param() 这个方法，你需要使用 req.params 来代替，注意这是一个对象，而不是方法❷。和 Express 类似，传入整数来调用 res.send() 会返回状态码给客户端❸。

使用其他的 HTTP 头部

在这一节介绍的技术中，你已经看到了如何使用 Content-Type 和 Accept 头部来处理不同的数据格式。还有其他一些有用的头部，在构建 APIs 时应该了解一下。

其中有一个是 restify 支持的 Accept-Version。当你在定义路由时，可以把一些选项包含在第一个参数中，而不是使用一个字符串。version 属性允许你的 API 根据 Accept-Version 来做出不同的响应。

例如, 使用 `app.get({ path: '/pages', version: '1.1.8' }, routes.v1.pages);` 这可以让你给版本 1.1.8 绑定一个特定的路由处理方法。如果不得不在 2.0.0 版本修改 API 时, 那么可以直接处理而不会影响旧版本的客户端。

没有什么能够阻止你在 Express 应用程序中使用这个头部信息, 但是在 `restify` 中更加容易使用。如果决定采用这种方式, 你应该学习一下如何使用 `major.minor.patch` 来处理语义化版本 (<http://semver.org>)。

如果你下载了整个例子 (`listings/web/restify`) 并且运行, 可以尝试我们之前介绍的一些 Curl 命令。创建、更新和展示, 应该都可以以同样的方式运行。

Express 和 `restify` 应用是相似的, 了解这一点是很有用的, 因为你可以开始组合两个框架所构建的应用。这两者都是基于 Node 的 `http` 模块, 这意味着你可以使用 `app.use(restify-App)` 在 Express 中挂载一个 `restify` 的应用。如果 `restify` 应用是单独的一个模块, 这种方式是非常好的——可以使用 `npm` 来安装, 然后放在单独的目录中。

Express 和 `restify` 都使用中间件, 你会发现结构良好的应用都是由多个松散耦合的中间件组成的, 这些中间件可以在不同的项目中重复使用。在下一节技巧内容中, 你将了解到如何编写自己的中间件, 这样便可以开始使用自定义日志等有用的功能来扩展应用程序。

技巧 72 使用自定义的中间件

你已经见过用于处理错误的中间件, 也已经在使用 Express 内置的中间件。你还可以使用中间件来添加自定义的路由、添加新的功能、优化日志, 或者做基于身份或者权限验证的访问控制。

中间件的好处是能够在应用程序中提高代码的复用程序。这一节内容会教你如何编写自己的中间件, 这样便可以在多个项目之间共享代码, 并且以更具可读性的方式来组织项目结构。

问题

你想要以可复用、可测试的方式来添加一些行为, 并且可以在访问特定的路由时触发这些行为。

解决方案

编写自己的中间件。

讨论

当你第一次开始使用 Express，中间件听起来像是一个复杂的概念，其他开发者用它来编写插件扩展 Express。但事实上，编写中间件是 Express 的基本组成部分，你应该尽快开始中间件的开发。如果你写过路由，那么便能够编写中间件：它们基本上使用了同样的 API！

在技巧 70 中，你看到了如何使用中间件来处理错误。错误处理是一个特殊的例子——你不得使用四个参数来获取错误对象：`app.use(function(err, req, res, next)) {`。对于其他中间件，你可以只使用三个参数，就像标准的路由处理器一样。这里有一个最简单的中间件：

```
app.use(function(req, res, next) {  
  console.log('%s %s', req.method, req.url);  
  next();  
});
```

❶

❷

❶ 调用 `app.use()` 来应用一个中间件。

❷ 调用 `next()` 继续执行下一个中间件。

传递一个匿名函数给 `app.use` ❶，这样的中间件通常都会执行，除非上一个中间件调用 `next` 失败。当你的代码执行完成后，你可以调用 `next` ❷ 来触发堆栈中的下一个中间件。这意味着两个事情：异步的 APIs 是可以支持的，并且添加中间件的顺序是非常重要的。

下面的实例会演示如何在中间件里边使用异步的 APIs。这个例子是在 `session` 中已经设置了用户 ID 的前提下，来加载一个用户信息。

```
app.use(function(req, res, next) {  
  if (req.session.user_id) {  
    db.users.find(req.session.user_id, function(err, user) {  
      if (err) {  
        next(err);  
      } else if (user) {  
        res.locals.user = user;  
        next();  
      } else {  
        next(new Error('Account not found'));  
      }  
    });  
  } else {  
    next();  
  }  
});
```

❶

❷

❸

❹

- ❶ 这个回调在每一次请求都会执行。
- ❷ 如果用户 ID 已经在 session 中设置了，那么加载这个账号的信息。
- ❸ 如果加载用户出现错误，把错误传递给错误处理的中间件。
- ❹ 如果用户已经加载了，在 res.locals 中设置，这样其他地方也可以使用。

这个中间件会在每一个请求中被触发❶。它会从数据库中加载用户账户信息，但仅限于当前用户的 ID 已经在 session 中设置了❷。加载用户的这个代码是异步的，所以 next 会在些许延迟后被调用。这里有几处的 next 调用：例如，如果在加载用户中出现错误，调用 next 时传回一个错误对象❸。

在这个例子中，加载的用户作为一个属性设置在 res.locals 中❹。使用 res.locals，你可以在其他中间件、路由处理器或者模板中访问这个用户信息。

这并不是使用中间件的最好方式。在这里，有一个异步函数意味着测试比较困难——你只能通过启动整个 Express 应用来测试中间件。你可能想要编写一些简单的单元测试，而不依赖于 HTTP 请求，所以最好是把代码重构成一个函数。这个函数有同样的声明，可以这样来使用：

```
var middleware = require('./middleware');  
app.use(middleware.loadUser);
```

- ❶ 加载一个包含中间件的模块。

所有的中间件都组织起来作为模块❶，你可以在其他位置加载这些中间件，即使是在完全不同的项目，或者测试代码，或者分离的路由中。这个函数可以解耦中间件来提高它的复用性。

如果你已经在使用技巧 67 介绍的路由分离模式，那么这会更有意义，因为这样中间件可以在不同文件定义的特定路由中使用。假设你正在使用技巧 71 中提到的 RESTful 的 API 风格，你的页面资源仅允许已经登录的用户更新，但是应用的其他部分允许任意用户访问，可以像这样来控制页面资源路由的访问权限：

```
var middleware = require('./middleware');  
  
app.get('/pages', routes.pages.index);  
app.get('/pages/:id', routes.pages.show);  
app.post('/pages', middleware.loadUser, routes.pages.create);  
app.patch('/pages/:id', middleware.loadUser, routes.pages.patch);
```

- ❶ 任何人都可以查看页面。
- ❷ 只有登录的用户可以创建或者更新页面。

在这个代码片段中，路由以一种称为页面的资源来进行定义。一些路由是任何人可以访问的❶，但是创建和更新页面仅限于系统内有账户的人们❷。在定义路由时，把 `loadUser` 中间件作为第二个参数传递便可以了。事实上，可以使用多个参数——你可以有一个普通用户加载的路由，和有特殊权限检查的路由来确保用户是管理员，或者有更改页面的必要权限。

图 9.4 展示了请求是如何在多个回调函数中传递，直到最后的请求发送到客户端的。有时候，有可能响应已经处理了，但是其他的中间件还没有执行——如果出现错误并且使用 `next(err)` 来传递。

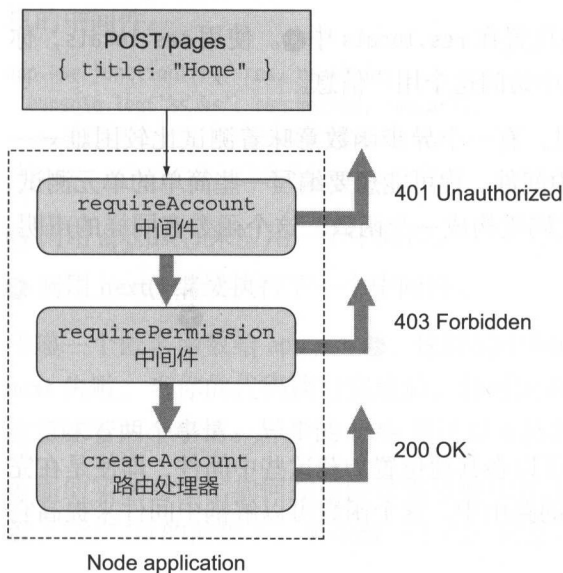


图 9.4 请求可以在多个回调函数中传递，直到响应最后被发送出去

你甚至可以一次性在多个路由中应用中间件。这在 Express 应用是很常见的，类似：`app.all('/admin/*', middleware.loadUser);`

如果使用 Node 的模块来管理中间件，把共享的功能独立文件存放来简化路由处理器，然后你就会发现把中间件组织成模块是组织应用的基本架构工具。

如果你正在设计一个新的 Express 应用，应该在中间件的方面去考虑。询问一下自己哪些 HTTP 请求需要去处理，以及它们可能需要哪种过滤。

现在是时候把这些想法整理成一个可以运行的例子。例子 9.22 展示了解析包含有 XML 请求的一种方式。中间件用于解析 XML，把它转换成一个纯粹的 JavaScript 对象。这意

意味着两件事情：你的代码只有少部分需要关系 XML，并且你也可以继续添加其他数据格式的支持。

例子 9.22 三种类型的中间件

```
var express = require('express');
var app = express();
var Schema = require('validate');
var xml2json = require('xml2json');
var util = require('util');
var Page = new Schema();

Page.path('title').type('string').required();

function ValidatorError(errors) {
  this.statusCode = 400;
  this.message = errors.join(', ');
}
util.inherits(ValidatorError, Error);

function xmlMiddleware(req, res, next) {
  if (!req.is('xml')) return next();

  var body = '';
  req.on('data', function(str) {
    body += str;
  });

  req.on('end', function() {
    req.body = xml2json.toJson(body.toString(), {
      object: true,
      sanitize: false
    });
    next();
  });
}

function checkValidXml(req, res, next) {
  var page = Page.validate(req.body.page);
  if (page.errors.length) {
    next(new ValidatorError(page.errors));
  } else {
    next();
  }
}
```

```
function errorHandler(err, req, res, next) {  
  console.error('errorHandler', err);  
  res.send(err.statusCode || 500, err.message);  
}  
  
app.use(xmlMiddleware);  
  
app.post('/pages', checkValidXml, function(req, res) {  
  console.log('Valid page:', req.body.page);  
  res.send(req.body);  
});  
  
app.use(errorHandler);  
  
app.listen(3000);
```

- ❶ 定义数据的校验来确保页面是有标题的。
- ❷ 从标准的错误对象中继承，所以校验出现的错误可以在错误中间件中处理。
- ❸ 这个函数将用于处理 XML 的中间件。
- ❹ 当 body 从客户端读取到的时候，请求对象将触发 data 事件。
- ❺ 数据校验的中间件。
- ❻ 传递错误给 next() 将阻止路由继续运行。
- ❼ 这是错误处理的中间件。
- ❽ 应用 XML 中间件到所有的请求中去。
- ❾ 特定的请求校验 XML。
- ❿ 添加最后一个中间件用于处理错误。

概括起来，这个例子定义了三个中间件来解析、校验 XML，然后响应一个 JSON 对象或者展示一个错误。我们在这里使用了一个任意类型的数据格式校验类库❶——你的数据库模块可能也有类似的功能。

这些路由处理 page 资源，并且期望这些 page 的格式是 XML 的，它会作为请求的 body 传递进来并且进行校验。一个错误对象，ValidatorError❷，当不可用的数据发送给服务器时用来返回一个 400 错误。XML 解析器❸会使用标准的基于事件的 API 来从请求的 body 中读取数据❹。这个中间件会于每一个请求到来时调用❽，因为它是直接传递给 app.use 的，但是它仅在当 Content-Type 设置为 XML 时执行。

数据校验的中间件❺会确保已经设置了页面的标题——这是我们选择的随意一个例子

来说明这种校验是如何工作的。如果数据没通过校验，那么一个 `ValidatorError` 的实例会传递给 `next` ❹。这将会触发错误处理的中间件 ❺。

数据仅会在特定的请求中校验。这是在 `/pages` 路由定义时传递 `checkValidXml` 方法来处理的 ❻。

全局的错误处理器是最后一个添加的中间件 ❼。应该始终都是这样的，因为中间件的执行是按照它定义的顺序来的。当 `res.send` 调用时，没有其他的处理程序会被触发，所以错误也不会抛出。

来测试这个例子，执行 `node server.js` 然后尝试使用 `curl` 来传一个 XML 给服务器：

```
curl -H "Content-Type: application/xml" \
  -X POST -d '<page><title>Node in Practice</title></page>' \
  http://localhost:3000/pages
```

你应该试一下不携带 `title` 来确保这样会抛出一个 400 错误。

这种方法可以用于 XML、JSON、CSV，或者其他任意你喜欢的数据格式。它非常适用于最小化的代码来处理 XML，但是还有其他方法在 Node web 应用中编写解耦的代码。在下一个技巧中，你将会看到对于 Node 来说很重要的东西——事件——可以作为另外一种很有用的架构模式来使用。

技巧 73 使用事件进行解耦

在大部分的 Express 应用中，很多代码都是使用方法和模块来进行组织的。在某些情况下，功能的共享不是特别方便，特别是你需要在应用中分离出关注的内容时。这一部分内容会把发送电子邮件来作为一个不适合使用路由、模块和视图来进行抽象的例子。事件可以把电子邮件从路由中解耦出来，让发送电子邮件相关的代码和 HTTP 请求的代码保持距离。

问题

你想要处理一些和 HTTP 无关的事情，类似发送电子邮件，但不了解如何组织代码来降低耦合和方便测试。

解决方案

使用内置的 `EventEmitter` 对象，就像 `Express app` 对象一样。

讨论

Express 和 `restify` 应用遵从模型-视图-控制器（MVC）模式。模型通常用于保存数据，控

制器是路由对应的处理，视图是模板，通常存放在 view 目录。

有一些代码并不大适合这种分类。例如，你应该在哪里处理发送电子邮件的代码？电子邮件的生成并不属于某个路由，因为电子邮件和 HTTP 没有关系。但是像路由处理器，它需要模板。它也没有一个真正的数据模型，因为它无须和数据库进行交互。

我应该怎样在模块中处理发送电子邮件的代码？在这个例子中，当你创建一个新的账号，要发送电子邮件时，会初始化一个用户模型。你可以把电子邮件相关的处理代码写在 `User.prototype.registerUser` 这个方法中。问题是并不是每一次创建一个用户都是需要发送邮件的。可能在测试的时候会不大方便，而且需要周期性的维护。

为什么发送电子邮件不是特别适合数据模型或者 HTTP 路由，可以通过了解 SOLID 原则来尝试理解 (<http://en.wikipedia.org/wiki/SOLID>)。这里有两个原则有关：单一职责和依赖倒置。

单一职责意味着处理 HTTP 路由的类真的不应该发送电子邮件，因为这是完全不同的两个工作，不能混在一起。控制反转是依赖倒置的一种方式，可以移除直接的调用——不使用类似 `emails.sendAccountCreation` 的方法，你的发送电子邮件相关的类应该只响应相关的事件。

对于 Node 开发者来说，事件是最重要的使用工具之一。幸运的是，SOLID 原则告知我们把电子邮件相关的代码从 HTTP 路由的处理代码中移出来，改成使用抽象的事件方式来实现，来让代码变得更好。这些事件可以对相应的类来做出响应。

图 9.5 展示了我们的应用结构应该是怎么样的。但是如何实现？例如 Express 应用，并没有一个合适的全局事件对象。你可以在某个地方创建一个全局变量，就像调用 `express()` 方法的文件一样，但是这样会暴露一个全局可共享的状态，这样会破坏我们之前提及的原则。

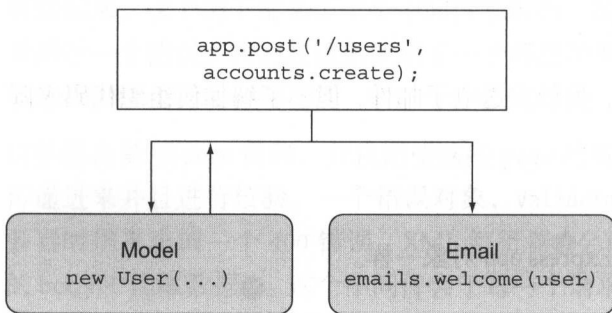


图 9.5 如果根据 SOLID 原则来进行组织，应用会更容易被理解

幸运的是，Express 已经在请求中有一个引用指向应用对象。路由的处理中，当拿到 req、res 参数时，可以使用 res.app 来获取应用对象。这个对象继承 EventEmitter，所以当需要时可以使用它来进行事件广播。如果你在路由处理中创建和保持了新的用户，它可以触发一个 res.app.emit('user:created', user) 或者类似的事件——你可以使用任何符合需要的命名。然后可以监听 user:created 事件来根据需要进行响应处理。这里可以包含发送电子邮件的通知，也可以做用户相关的日志统计。

下边的代码会展示如何在应用对象上监听事件。

例子 9.23 使用事件来组织应用结构

```
var express = require('express');
var app = express();
var emails = require('./emails');
var routes = require('./routes');

app.use(express.json());

app.post('/users', routes.users.create);

app.on('user:created', emails.welcome);

module.exports = app;
```

①

②

① 设置一个路由来创建用户。

② 监听创建用户的事件，来绑定到对应的 email 代码。

在这个例子定义了一个注册用户的路由①，然后定义了一个事件监听器，绑定一个发送 email 的方法②。

这个路由可以在下边的例子看到。

例子 9.24 触发事件

```
var User = require('../models/user');

module.exports.create = function(req, res, next) {
  var user = new User(req.body);
  user.save(function(err) {
    if (err) return next(err);
    res.app.emit('user:created', user);
    res.send('User created');
  });
};
```

①

❶ 当用户成功注册时触发创建用户的事件。

上述代码包括了一个用户对象的数据模型例子。如果成功创建用户了，那么 `user:created` 事件会在应用对象上触发。下载本书相关的源码可以获得发送电子邮件的完整例子，但是移除直接调用和坚持单一职责的基础原则的方法已经在这里展示了。

在应用中使用事件进行通信是十分有用的，特别是当你需要让代码对其他开发者来说更容易理解时，或者需要多次和客户端代码进行数据交换时。下一个技巧将会为你展示在 Node 应用中如何利用 WebSockets，同时也会涉及如何获取类似 session 这样的数据。

技巧 74 使用 WebSockets 来处理 sessions

Node 现在对实时的 web 应用支持很好。使用面向事件的、异步的 APIs 意味着支持 WebSockets 有着天然的优势。同时，对 Node 来说，在一个进程中运行两个服务器也是毫无问题的：一个用于 WebSockets 的连接，一个用于标准的 Node HTTP 服务。

这个技术将会为你介绍如何复用 Connect 和 Express 中的中间件来创建一个 WebSocket 的服务。如果你的应用允许用户进行登录，那么需要在添加 WebSocket 支持的同时，了解如何在 WebSockets 中使用 session。

问题

你想要在已有的 Express 应用中添加 WebSocket 的支持，但是你不了解如何获取 session 变量，如用户当前是否已经登录。

解决方案

在你的 WebSocket 服务中复用 Connect 中的 cookie 和 session 中间件。

讨论

这一节假设你已经对 WebSockets 有一定的了解了。回顾一下：HTTP 请求是无状态的，相对时间较短，在下载文件，或者请求资源的状态改变上是很棒的。但是如果是来自服务器的数据流呢？

某些类型的事件是由服务器发起的。想想一个 web 邮件服务。当你创建和发送邮件时，你推送到服务器，服务器将其发送给收件人。如果收件人正看着他们的收件箱，有没有简单的方法让他们的浏览器来获取更新。它可以使用 Ajax 来进行检查请求新的消息，但是这个实现不是特别优雅。服务器知道它有一个新的消息收件人，如果它可以直接推送数据信息给用户，那会是更好的方式。

这就是 WebSockets 的由来。在概念上更像在第 7 章看到的 TCP socket：一个在客户端

和服务器之间双向的桥梁。要实现这一点，除了标准的 Express 服务器，或者单纯的旧的 Node 服务器之外，还需要一个 WebSockets 的服务器。图 9.6 中解释了在普通的 Node web 应用中是如何工作的。

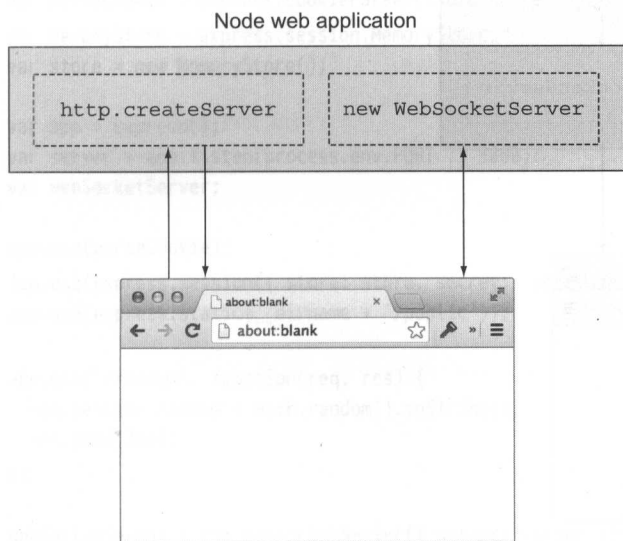


图 9.6 一个 Node 的 Web 应用应该同时支持标准的 HTTP 请求和 WebSockets

HTTP 请求是短暂的，有特定的端点，并使用类似 POST 和 PUT 的方法。WebSockets 是常驻的，且没有特定端点，没有方法。两者在概念上差异比较大，但由于它们使用相同的应用程序进行通信，通常需要访问相同的数据。

这说明有一个 sessions 问题。我们已经看过的 Express 例子使用中间件来自动加载 session。Connect 中间件是基于 HTTP 请求和响应的，那么我们如何把这个应用到常驻以及双向的 WebSockets 上。为了帮助我们理解这个，需要看下 WebSockets 是如何工作的。

Sessions 是基于 cookies 中的唯一标识来进行加载的。WebSockets 使用标准的 HTTP 请求来进行初始化，然后升级为 WebSocket 的连接。这意味着有一个时间点你可以把 cookie 从请求中获取，然后加载 session。对于每一个 WebSocket，你可以存储一个用户 session 的引用。现在你可以做需要 session 时一样的事情：验证用户是否已经登录，设置引用即可。

图 9.7 在图 9.6 的基础上来展示在 WebSockets 中如何使用 sessions，通过使用 Connect 中间件来解析 cookies 以及加载 session。

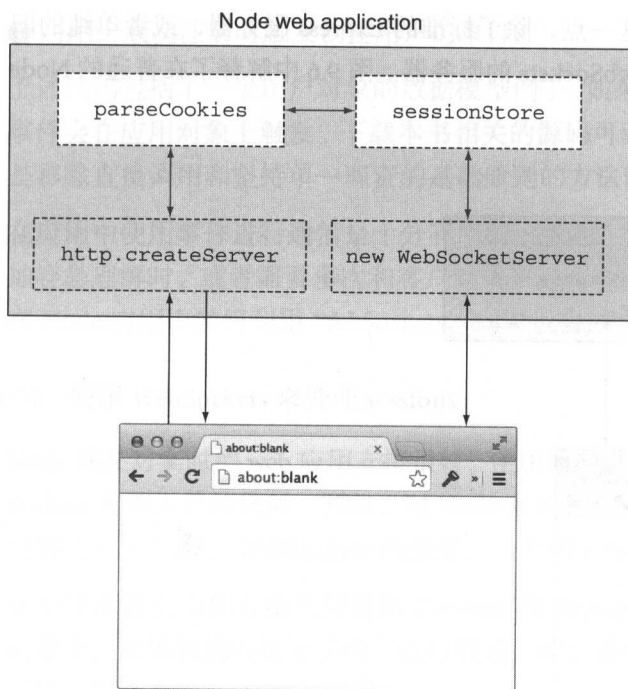


图 9.7 从 WebSockets 访问 sessions 信息

现在你知道这一部分如何很好地结合在一起，你应该如何构建？cookie 解析的中间件可以在 `express.cookieParser` 找到。有一个简单的方法来从请求头部信息获取 cookie，然后将 cookie 的字符串解析成分离的值。它可以接受一个参数，`secret`，用于给 cookie 来进行签名的值。一旦 cookie 解密好了，你可以获取到 session ID 并且加载 session。

在 Express 中的 sessions 是基于异步的 API 来进行存储和获取的。它们可以使用数据库来进行存储，或者使用内置的基于内存的类。如果 session ID 是正确的，把 session ID 和回调给 `sessionStore.get` 便可以加载 session。

在这一节内容中我们使用 WebSocket 模块 `ws` (<https://www.npmjs.org/package/ws>)。这是一个快捷精简的实现，和 Socket.IO 有着很不一样的 API。如果你想要学习 Socket.IO 相关的，那么 *Node in Action* 中有很优秀的一个教程。这里我们使用更加简单的模块来帮助你理解 WebSockets 是如何工作的。

使用 `ws` 来加载 session，你需要从 HTTP 初始化请求中解析 cookies，然后调用 `sessionStore.get`。一个完整的例子会展示是如何运作的。

例子 9.25 使用 WebSockets 的 Express 应用

```
var express = require('express');
var WebSocketServer = require('ws').Server;
var parseCookie = express.cookieParser('some secret');
var MemoryStore = express.session.MemoryStore;
var store = new MemoryStore();

var app = express();
var server = app.listen(process.env.PORT || 3000);
var websocketServer;

app.use(parseCookie);
app.use(express.session({ store: store, secret: 'some secret' }));
app.use(express.static(__dirname + '/public'));

app.get('/random', function(req, res) {
  req.session.random = Math.random().toString();
  res.send(200);
});

websocketServer = new WebSocketServer({ server: server });

websocketServer.on('connection', function(ws) {
  var session;

  ws.on('message', function(data, flags) {
    var message = JSON.parse(data);

    if (message.type === 'getSession') {
      parseCookie(ws.upgradeReq, null, function(err) {
        var sid = ws.upgradeReq.signedCookies['connect.sid'];

        store.get(sid, function(err, loadedSession) {
          if (err) console.error(err);
          session = loadedSession;
          ws.send('session.random: ' + session.random, {
            mask: false
          });
        });
      });
    } else {
      ws.send('Unknown command');
    }
  });
});
```

- ❶ 加载解析 cookie 的中间件和设置密码。
- ❷ 加载要使用的会话存储。
- ❸ 告知 Express 使用会话存储和设置密码。
- ❹ 创建 Express 路由来设置测试用的会话值。
- ❺ 设置 WebSocket 服务器，将其传给 Express 服务器。
- ❻ 在连接事件中，给客户端创建 WebSocket。
- ❼ 客户端发送的数据假设是 JSON 格式，在这里进行解析。
- ❽ 从 HTTP 的更新请求中获取 WebSocket 的会话 ID。
- ❾ 从存储中获取用户的会话信息。
- ❿ 从会话中通过 WebSocket 把值发送回去。

这个例子从加载和配置 cookie 解析器❶和 session 存储❷开始。我们使用签名的 cookies，所以留意下，`ws.upgradeReq.signedCookies` 在加载 session 之后会使用到。

Express 已经设置使用 session 中间件❸，我们已经创建了一个路由，你可以用于测试❹。在浏览器打开 `http://localhost:3000/random` 在 session 中设置一个随机的值，然后访问 `http://localhost:3000/` 来查看打印了什么。

`ws` 模块使用一个原生的构造器，`WebSocketServer`，来处理 WebSockets。为了使用它，你要使用 Node HTTP 服务器对象来进行初始化——在这里传递已有的 Express 服务就可以了❺。当服务器开启时，连接创建时它便会触发相应的事件❻。

例子中的客户端发送一个 JSON 给服务器，所以这里需要一些代码来解析 JSON 字符串确定是否可用❼。在这个例子中没必要完整地展示出来，但是我们将其囊括了进来，表明 `ws` 需要这些另外的工作，这也可以在大多数实际情况下使用。

一旦 WebSockets 服务器有了一个连接，session ID 可以从初始化请求中的 cookies 中获取❽。这和 Express 中是很相似的——我们只是需要在初始化的请求中传递一个引用给解析 cookie 的中间件。然后 session 可以使用 session 存储的 `get` 方法来加载❾。session 加载后，会把一个包含了 session 中的值的消息发送回客户端❿。

在这个例子中，客户端需要运行的代码实现如下所示。

例子 9.26 客户端的 WebSocket 实现

```
<!DOCTYPE html>
<html>
<head>
```

```
<script>
var host = window.document.location.host.replace(/:.*/, '');
var ws = new WebSocket('ws://' + host + ':3000');

setInterval(function() {
  ws.send('{ "type": "getSession" }');
}, 1000);

ws.onmessage = function(event) {
  document.getElementById('message').innerHTML = event.data;
};
</script>
</head>
<body>
  <h1>WebSocket sessions</h1>
  <div id='message'></div><br>
</body>
</html>
```

❶

❶ 定期向服务器发送消息。

这部分所做的事情就是发送一个消息给服务器端。它会显示 `undefined`，除非你访问了 `http://localhost:3000/random`。如果你打开两个窗口，一个是 `http://localhost:3000/random`，另外一个 `http://localhost:3000/`，你可以刷新 `random` 页面，然后 `WebSocket` 页面会显示新的值。

运行这个例子需要 `Express 3` 和 `ws 0.5`——我们已经准备了一个书中例子都可以使用的 `package.json`。

下一节技术内容会有关于从 `Express 3` 迁移到 `Express 4` 的相关提示。

技巧 75 升级 Express 3 到 4

这本书是在 `Express 4` 发布之前写的，所以我们的 `Express` 例子都是使用这个框架的版本 3 来编写的。我们也在这一章介绍一下升级 `Express` 的相关技术，你也可以看一下版本 4 和旧的版本上的一些差异。

问题

你有一个使用 `Express 3` 的应用需要升级到 `Express 4`。

解决方案

升级应用的配置，安装缺失的中间件，使用新的路由 API。

讨论

Express 3 到 4 的更新是一个很长的时间了。某些更改已经暗示在 Express 3 的文档，因此 API 的变化不是非预期的，甚至过于戏剧化的大部分。你可能会花大部分的时间更换 Express 中间件，因为 Express 4 不再有任何内置的中间件组件，除了 `express.static`。

该 `express.static` 中间件组件能够快速安装包含的 JavaScript、CSS 和图像资源的 `public` 文件夹。这一直留着，因为它的方便，但中间件组件的其余部分都没有了。这意味着你将需要 `npm install --save body-parser`（如果以前使用 `bodyParser`）。请参考表 9.3 具有旧中间件的名称和较新的相同的模块组件。只要记住，你需要 `npm install --save` 每一个，然后在你的 `app.js` 文件使用就可以了。

表 9.3 集成 Express 中间件

Express 3	Express 4 npm package	Description
<code>bodyParser</code>	<code>body-parser</code>	解析 URL 编码和 JSON POST 的请求 body 数据
<code>compress</code>	<code>compression</code>	压缩服务器的响应
<code>timeout</code>	<code>connect-timeout</code>	如果请求花费时间太多，允许超时
<code>cookieParser</code>	<code>cookie-parser</code>	从 HTTP 头部信息中解析 cookies，结果存放在 <code>req.cookies</code>
<code>cookieSession</code>	<code>cookie-session</code>	使用 cookies 来支持简单的会话
<code>csrf</code>	<code>csurf</code>	在会话中添加 token，你可以使用这个来防御 CSRF 攻击
<code>error-handler</code>	<code>errorhandler</code>	Connect 中使用的默认错误处理
<code>session</code>	<code>express-session</code>	简单的会话处理，使用 <code>stores</code> 扩展来把会话信息写入到数据库或者文件中
<code>method-override</code>	<code>method-override</code>	映射新的 HTTP 动词到请求变量中的 <code>_method</code>
<code>logger</code>	<code>morgan</code>	日志格式化
<code>response-time</code>	<code>response-time</code>	跟踪响应时间
<code>favicon</code>	<code>serve-favicon</code>	发送网站图标，如果没有的话默认会使用一个内置的
<code>directory</code>	<code>serve-index</code>	目录列表，和 Apache 的目录索引类似
<code>vhost</code>	<code>vhost</code>	允许路由匹配子域名

你可能不使用这些模块。在我的应用程序 I（Alex）中通常只有 `body-parser`、`cookie-parser`、`csurf`、`express-session` 和 `method-override`，所以迁移并不太难。下面列出了使用这些中间件组件的小应用。

例子 9.27 Express 4 中间件

```
var bodyParser = require('body-parser');  
var cookieParser = require('cookie-parser');  
var csrf = require('csrf');  
var session = require('express-session');  
var methodOverride = require('method-override');  
var express = require('express');  
var app = express();
```

```
app.use(cookieParser('secret'));  
app.use(session({ secret: 'secret' }));  
app.use(bodyParser());  
app.use(methodOverride());  
app.use(csrf());
```

```
app.get('/', function(req, res) {  
  res.send('Hello');  
});
```

```
app.listen(3000);
```

❶ 加载中间件模块。

❷ 配置每一个中间件。

❸ 定义路由。

要安装 Express 4 和这些必要的中间件，你应该在新的目录下执行以下命令：

```
npm install --save body-parser cookie-parser \  
  csrf express-session method-override \  
  serve-favicon express
```

这样会安装所有独立于 Express 4 的依赖模块，并且把它们的信息保存到 package.json 文件中。一旦使用 require 来加载时❶，你可以使用 app.use 来将它们添加到应用的栈中，如同在 Express 3 所做的一样❷。路由的处理器也可以像在 Express 3 中一样添加❸。

官方的迁移指南

Express 的作者在 Github 的 Express 项目的 wiki 中编写了迁移的指南文档，^a包括了每一个变化的简单介绍。

^a<https://github.com/visionmedia/express/wiki/Migrating-{}from-{}3.x-{}to-{}4.x>

你没法再使用 `app.configure`，但是停止使用是很简单的一件事情。如果你使用 `app.configure` 用于配置特定的环境，那么可以使用 `process.env.NODE_ENV` 来做一个条件判断。下边的例子是一个假设的 `logger` 中间件，在测试环境中并不希望打印详细的日志信息。

```
if (process.env.NODE_ENV !== 'test') {  
  app.use(logger({ verbose: true }));  
}
```

新的路由 API 强化了小型应用程序可以划分在不同的模块的概念。意味着你的 RESTful 资源可以脱离资源的名称，不是写 `app.get('/songs', songs.index)`，现在你可以这么写了：`songs.get('/', index)`，然后把 `songs` 使用 `app.use` 加载。这和技巧 67 中提到的路由分离的概念相当符合。

下边的例子会为你展示如何使用新的路由 API。

例子 9.28 Express 4 中间件

```
var express = require('express');  
var app = express();
```

```
app.get('/', function(req, res) {  
  res.send('Hello');  
});
```

```
var songs = express.Router();
```

```
songs.get('/', function(req, res) {  
  res.send('A list of songs');  
});
```

```
songs.get('/:id', function(req, res) {  
  res.send('A specific song');  
});
```

```
app.use('/songs', songs);
```

```
app.listen(3000);
```

❶ 创建新的路由。

❷ 添加路由处理方法到这个路由集合中。

❸ 使用 URL 前缀来挂载路由。

在创建了一个新的路由之后❶，你可以如你熟悉的方式添加路由，例如 HTTP 的动作

get ②。最赞的是你也可以添加中间件到这一部分的路由匹配中，只需要调用 `songs.use`。这在老版本的 Express 会是相当麻烦的一件事情。

一旦你设置了一个路由，可以使用 URL 前缀来进行挂载③，这意味着可以把相同的路由挂载到不同的 URLs 上，很容易地对其进行重命名。

如果你把路由放在独立的文件中，然后在主要入口 `app.js` 文件中去挂载，那么可以把路由作为一个模块分发到 `npm` 中去。这意味你可以在多个应用中复用路由。

最后必须要提到的关于 Express 4 的一件事情是新的 `router.param` 方法。这允许你在确定路由参数是可以使用异步的代码。假设你有一个 `"/songs/:song_id"` 和 `:song_id` 应该是在数据库中的一个可用的歌曲 ID。使用 `route.param` 方法，你可以在路由处理器执行之前，验证该值是否是一个数值变量，并且在数据库中是存在的！

```
router.param('song_id', function(req, res, next, id) {
  Song.find(id, function(err, song) {
    if (err) {
      return next(err);
    } else if (!song) {
      return next(new Error('Song not found'));
    }
    req.song = song;
    next();
  });
});

router.get('/songs/:song_id', function(req, res, next) {
  res.send(req.song);
});
```

在这个例子中，`Song` 假设是一个从数据中获取歌曲数据的类。真正处理 `router` 的内容是比较简单的，因为它只在当可用的歌曲找到时才运行。另外一点，`next` 可以将异常错误传递给错误处理的中间件。

上述内容已经包括了 web 应用程序开发技术部分。在一个更重要的事情之前，我们会先介绍下一节内容。和其他事物一样，web 应用程序应该需要很好的测试。下一节将会介绍测试 web 应用程序已经可以使用的一些有用技巧。

9.3 web 应用程序的测试

测试看起来就是一个苦差事，但是它也是用于验证想法不可缺少的工具，尤其是你创建没有用户界面的 web APIs 时，更需要测试。

第 10 章会介绍在 Node 中的测试，技巧 84 会有一个关于测试 web 应用程序的例子。下一个技巧我们会扩展这个例子来给你展示如何测试登录验证的路由。

技巧 76 测试路由

测试框架，例如 Mocha 让测试用例更加容易编写和阅读，SuperTest 则用于帮助保持 HTTP 相关的测试的简洁。但是身份认证的支持通常并没有类似的模块来处理。在这一节的内容中，你可以了解到一种在测试中处理身份认证的方法，这已经足够使用，并且也可以在其他测试模块中去复用。

问题

你想要测试应用中基于 session 的用户名和密码的那一部分内容。

解决方案

在测试用户的 setup 中发起请求进行登录，然后在测试用例中复用这个 cookies 信息。

讨论

一些 web 框架和测试库会处理会话，所以你可能没有太担心登录。在这本书之前的内容使用过的 Mocha 和 Super 中可不是这样，所以你需要大概了解了一下 session 是如何工作的。

Express 连接的 session 处理是基于一个 cookie。一旦该 cookie 已经设置，它可以被用来读取用户的 session。这意味着，编写一个测试，访问你的应用程序的安全部分，你需要做的请求，在用户登录后记录 cookie，然后在随后的请求中附带这个 cookie。这个过程示于图 9.8。

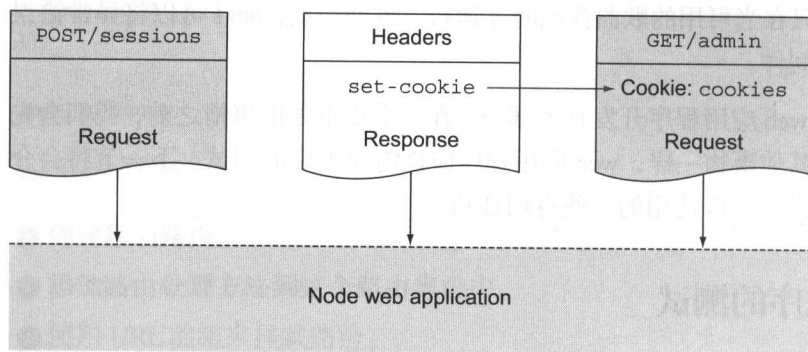


图 9.8 你可以通过捕获 cookies 来测试身份验证的路由

编写访问身份验证的路由测试，你需要一个测试用户账户，通常需要创建数据库数据。在第 10 章技巧 87 中可以了解相关内容。

一旦数据准备好了，你可以像使用 SuperTest 库做一个用户名和密码的 POST 请求到处理服务器上。cookie 的传输是使用 HTTP 头部信息，这样你就可以从 `res.headers['set-cookie']` 来获取。你也应该做一个断言，以确保账户登录成功。

现在，任何新的请求只需要在 `res.headers` 中设置 Cookie 信息，你的测试用户即是登录状态了。接下来的例子展示了这一过程。

例子 9.29 测试已经验证了的请求

```
var app = require('..../app');
var assert = require('assert');
var request = require('supertest');

var administrator = {
  username: 'admin',
  password: 'secret'
};

describe('authentication', function() {
  var cookies;

  before(function(done) {
    request(app)
      .post('/session')
      .field('username', administrator.username)
      .field('password', administrator.password)
      .end(function(err, res) {
        assert.equal(200, res.statusCode);
        cookies = res.headers['set-cookie'];
        done();
      });
  });

  it('should allow admins to access the admin area', function(done) {
    request(app)
      .get('/admin')
      .set('Cookie', cookies)
      .expect(200, done);
  });
});
```

❶ 这是一个测试用户，通常从一个固定的设置中加载。

- ❷ Post 提交用户名和密码。
- ❸ 会话 cookie 在 set-cookie 的头部信息中。
- ❹ 这个路由在登录之后才执行。
- ❺ 使用保存的会话 cookie 来设置 Cookie 头部信息。

本试验的第一部分加载所需的模块并且设置了一个示例用户❶。这通常将被存储在数据库中。接下来，会发起一个附带用户名和密码信息的 POST 请求❷。会话 cookie 会在头部信息的 set-cookie 字段中获取❸。

在登录后，访问特定的路由❹，设置上一次保存的 Cookie 头部信息❺。你会发现，请求处理时，用户是在正常登录的状态。

理解如何进行 session 相关的测试的诀窍在于明白 Connect 的 session 中间件是如何工作的。其他的中间件在测试中并不是这么简单的，所以下一节的内容将会介绍测试的 seams 的概念，来帮助你在测试过程中更好地操作中间件。

技巧 77 为中间件注入创建 seams

中间件很灵活，可以组合使用。模块化的方式使得基于 Connect 的应用开发起来很是愉快。但是中间件有一个缺点，就是难以测试。这一节的技巧是如何通过创建 seams 来处理相关的问题。

问题

你在使用中间件，这让应用测试变得比较麻烦。

解决方案

在测试持续过程中找出用于替换中间件的 seams。

讨论

在计算机中，seam 这个词通常是用来描述可以用于编辑修改源代码的位置。这个概念在 Stephen Vance 在他的书 *Quality Code: Software Testing Principles, Practices, and Patterns*³ 中被扩展应用到类似 JavaScript 的语言中。

在代码中 seam 在测试环境中提供了一种方式来控制代码执行。任何地方我们可以执行、覆盖、注入或者控制代码都可以是 seam。

³<https://www.informit.com/store/quality-code-software-testing-principles-practices-9780321832986>

这其中一个例子是源于 Connect 的中间件 csrf。它创建了一个 session 变量，可以在表单中引入来防止跨站请求攻击。你有一个 web 应用是允许注册用户创建日历项。如果你的网站没有使用 CSRF 保护，其他人可以创建一个 web 页面来欺骗你的用户删除日历中的内容。这个攻击看起来可能是这样的：

```

```

用户的浏览器还是会加载托管在外部网站上的图片。但是它以一种潜在危险的方式引用了你的网站。为了防止这种情况的发生，每一次请求中会随意产生一个 token 来插入到表单中。攻击者无法拿到 token，就无法进行有效攻击。

不幸的是，简单地添加 express.csrf 到渲染表单的路由并不能进行完整的测试。测试用例一定要经过表单的第一次加载才能提交 POST 请求到路由处理器中，而这过程中一定生成了包含 CSRF token 的 session 变量。

为了解决这个问题，你需要把 express.csrf 纳入控制之中。重构它来创建一个 seam：将其放在包含其他自定义中间件的模块中去，在测试过程中去修改它。你没有必要测试 express.csrf，因为 Express 和 Connect 的作者已经帮你做了测试——你需要的是在测试中改变它的行为。

有其他两个选择可以采用：检查 process.env.NODE_ENV 是否设置为 test，然后到仅用于测试版本的 CSRF 中间件分支，或者在 express.csrf 的内部打补丁来让你可以提取那个密码的 token。这两种方式都有问题：第一种方式意味着不可能有 100% 的代码覆盖率——你生产环境的代码不得不包括测试代码。第二种方式有可能很脆弱——如果 Connect 在未来修改了 CSRF 的功能就很可能有影响。

我们使用的基于 seam 的概念，要求你创建一个中间件文件，如果你没有。这只是一个文件用来把你所有的中间件都组织起来成为一个模块来方便加载。然后你需要创建一个方法把 express.csrf 给包裹起来，或者直接返回它。一个基础的例子如下所示。

例子 9.30 控制中间件

```
var express = require('express');
```

```
module.exports.csrf = express.csrf;
```

❶

❶ 创建一个用来注入其他中间件的地方。

这所做的一切就是把原始的 csrf 中间件给暴露出来❶，但是现在它在测试过程中注入不同的行为则要容易得多。下边的列表会展示一个测试是怎么样的。

例子 9.31 在测试中注入新的行为

```
var middleware = require('../middleware');

middleware.csrf = function() {
  return function(req, res, next) {
    req.session._csrf = '';
    next();
  };
};

var app = require('../app');
var request = require('supertest');

describe('calendar', function() {
  it('should allow us to turn off csrf', function(done) {
    request(app)
      .post('/calendars')
      .expect(200, done);
  });
});
```

- ❶ 首先加载中间件列表，然后替换它的 `csrf` 方法。
- ❷ 这里阻止设置 `_csrf` 的值。
- ❸ 应该返回 200，而不是 403。

这个测试在处理其他事情之前，先加载我们自定义的中间件模块，然后替换掉 `csrf` 方法❶。当它加载 `app` 然后使用 `SuperTest` 发起一个请求时，`Express` 会使用我们注入的中间件，因为 `middleware.js` 会被缓存起来。设置 `_csrf` 的值以防任何一个视图依赖它❷，然后请求应该是返回 200 而不是 403（`forbidden`）❸。

这看起来我们好像并没有做多少事情，但通过重构 `express.csrf` 加载的方式，我们可以以一种更方便测试的方式来允许我们应用。你通常可能会创建两个请求来确保 `csrf` 中间件的使用，但是这个技术也可以用来做其他的事情。如果在测试中有你不想运行的东西，找到 `seams` 来注入需要的行为，或者尝试使用简单的 `JavaScript` 或者 `Node` 模式来创建一个 `seam`——你不需要一个复杂的依赖注入框架，可以利用 `Node` 模块系统的优势。

下一个技术基于这些想法来让测试和模拟的远程服务进行交互。如果你在写一个访问远程服务的应用的测试，例如一个支付网关，这会帮助你更加轻松地完成。

技巧 78 测试依赖远程服务的应用

第三方模块可以让应用来集成远程服务，如 Github、Twitter 和 Facebook。但是你如何测试依赖于远程服务的应用？这一节会介绍如何在远程依赖中插入一些桩，来让测试变得更快，更易于维护。

问题

你正在使用一个社交网络进行身份验证，或者一个服务来接受支付，但你不希望测试用例来访问这些远程依赖。

解决方案

找出在你的应用、远程服务和那些你想要测试的东西之间的 seams，然后插入你自己的 HTTP 服务来模拟部分远程依赖。

讨论

大部分 web 应用需要的，但很容易有危险错误的东西是用户管理。使用 Node 模块来支持像 Github、Google、Facebook 和 Twitter 等公司提供的验证服务可能比构建一个定制的解决方案要更快且更加安全。

使用这些服务相对来说比较容易，但是你应该怎么测试呢？在技巧 76 中，你已经看到如何来给验证路由编写测试。这包括了登录后保存 session 和 cookies，随后的请求便已经是身份验证通过了。你不能对远程服务使用同样的方式，因为这样你的测试就不得不去访问生产环境真实的服务。你可以使用测试账号，但是如果你想要在离线的情況下执行测试呢？

为了解决这个问题，你需要在应用和远程服务之间创建一个 seam。每当你的应用程序尝试与远程服务进行通信时，你需要插入一个假的版本来触发类似的响应。在单元测试中，mock 对象可以用来模拟其他对象。你所需要的是就是 mock 一个服务。

对于你的应用，需要满足两个条件来让这一切变得可能：

- 可配置的远程服务。
- 可以提供远程服务的 web 服务器。

第一个条件意味着你的应用程序需要允许修改远程服务的 URL。如果需要连接到 `http://auth.example.com/signin`，那么你需要在测试的时候指定为 `http://localhost:3001/signin`。这个端口完全取决于你自己——我们已经见过一些解决方案在一次同样的测试中使用多个端口来运行多个服务。

不过第二个条件是可以满足的。如果你使用的是 Express，可以定义一组足够使用的路由和代码来启动 Express 服务器，模拟远程的服务。这个服务可以在它自己的模块中，在需要它的测试中加载进来。

在实践中，这并不需要太多的代码。所以一旦理解了这个原则，用它来处理大多数普通的 API 应该都不会特别困难。如果你正在尝试模拟的 API 没有良好的文档，那么可能需要捕获真实的请求来弄清楚它是如何工作的。

调查远程的 APIs

有些时候远程的 APIs 并没有良好的文档。当你需要比基础 API 调用更加深入时，必然会有一些不容易理解的部分。像这种情况，我们发现最好是使用一些命令行工具，如 curl，来创建请求，并且使用 HTTP 日志工具来监测请求和响应。

如果你在使用 Windows，Fiddler (<http://www.telerik.com/fiddler>) 是非常必要的。它是一个 HTTP 调试代理，同时也支持 HTTPS。

```
GET https://github.com/
  ↳ 200 text/html 5.52kB
GET https://a248.e.akamai.net/assets.github.com/stylesheets/bundles/github2-24f59e3ded11f2a1c7ef9ee730882bd8d550cfb8.css
  ↳ 200 text/css 28.27kB
GET https://a248.e.akamai.net/assets.github.com/images/modules/header/logov7@4x-hover.png?1324325424
  ↳ 200 image/png 6.01kB
GET https://a248.e.akamai.net/assets.github.com/javascripts/bundles/jquery-b2ca07cb3c906cecfd58811b430b8bc25245926.js
  ↳ 200 application/x-javascript 32.59kB
U GET https://a248.e.akamai.net/assets.github.com/stylesheets/bundles/github-cb564c47c51a14af1ae265d7ebab59c4e78b92cb.css
  ↳ 200 text/css 37.09kB
GET https://a248.e.akamai.net/assets.github.com/images/modules/home/logos/facebook.png?1324526958
  ↳ 200 image/png 5.55kB
>> GET https://github.com/twitter
```

[7] [i:.*]

?:help [*:8080]

Glance 有内置的错误页面

对于 Linux 和 Mac OS，mitmproxy (<http://mitmproxy.org/>) 是一个不错的选择。它允许实时监测 HTTP 传输，并且能够导出，保存请求信息和重发请求。我们发现

它很适合于调试基于 Node 用来支持桌面应用的 APIs，以及用来找出某些流行的支付网关的奇怪行为。

在下边的三个例子中，你可以看到如何创建一个 mock 服务器，可以在测试中用于模拟 Paypal 的行为。第一个例子展示的是应用本身的代码。

例子 9.32 使用 Paypal 的小型 web 商店

```
var express = require('express');
var app = express();
var PayPal = require('./paypal');
var paypal = new PayPal({
  user: 'NIP',
  paypalUrl: 'http://localhost:3001/validate',
  rootUrl: 'http://localhost:3000'
});
```

❶

```
app.use(express.bodyParser());
```

```
app.post('/buy', function(req, res, next) {
  var url = paypal.generateUrl(req.body);
```

❷

```
  // Send the user to the PayPal payment page
  res.redirect(url);
});
```

```
app.post('/paypal/success', function(req, res, next) {
  paypal.verify(req.body, function(err) {
    if (err) next(err);
    app.emit('purchase:accepted', req.body);
    res.send(200);
  });
});
```

❸

❹

```
module.exports = app;
```

❶ 这些设置用于控制 PayPal 的行为。

❷ 获取 PayPal 使用的支付 URL。

❸ 处理支付的通知信息。

❹ 当支付成功后，触发一个事件。

在靠近文件顶部的设置传递给 PayPal 类^❶用于控制 PayPal 的行为。其中一个，paypalUrl，可以是 <https://www.sandbox.paypal.com/cgi-bin/webscr> 来作为 PayPal 的临时服务器进行测试。在这里，我们要使用本地的 URL，因为我们要运行自己的一个 mock 服务器。

如果这是一个真正的项目，你应该使用一个配置文件来存储这些配置选项。然后，测试的配置可以指向本地服务器，也可以使用 PayPal 沙箱，真实环境使用 PayPal.com。要了解更多有关配置文件的内容，可以参考技巧 69。

要进行付款的时候，用户请求将会被转发到 PayPal 的托管服务上。我们演示中的 PayPal 类有生成该 URL 的作用，它会使用 paypalUrl^❷。这个例子还描述支付通知处理的功能实现^❸——在 PayPal 通常称为 IPN。

在这里添加的一个额外的东西是触发事件，emit^❹。这会使得它更容易测试，因为我们可以测试监听到 purchase:accepted 事件。它也可以用于设置单击邮件的处理，可以参考技巧 73 来了解更多内容。

现在 mock 的 PayPal 服务器，它所需要做的是处理 IPN 的请求。基本上它需要说，“是的，购买已经验证了。”它也可以选择性地报告错误，这样我们可以测试错误处理。下面的内容展示了一个简单的 mock 服务器的代码是怎么样的。

例子 9.33 模拟 PayPal 的 IPN 请求

```
var express = require('express');
var paypalApp = express();

paypalApp.returnInvalid = false;

paypalApp.post('/validate', function(req, res) {
  if (paypalApp.returnInvalid) {
    res.send('INVALID');
  } else {
    res.send('VERIFIED');
  }
});

module.exports = paypalApp;
```

❶ 错误报告是否开启。

❷ 处理 IPN 验证。

现实生活中，在销售快要完成时，PayPal 店会收到来自 PayPal 的 POST 请求查询订单的详细信息。你需要获取订单后将其发送回 PayPal 来进行验证。这可以防止攻击者创建一

个假的 POST 请求来欺骗应用创建一个假的购买订单。

这个例子包括了一个是否报告错误的开关^❶。我们不打算在这里使用，但是在实际项目中，如果你需要测试错误是如何被处理的，它非常有用。有些客户会遇到错误的情况，所以确保它们正常处理是非常重要的。

一旦所有的东西准备好了之后，我们需要做的就是发送回一个文本信息 VERIFIED^❷。这就是 PayPal 所做的一切，有时候却可以深奥得有点让人沮丧。

最后，我们看一下测试，把所有内容都整合到一起。下边的代码列表使用了 mock 的 PayPal 服务器和我们的应用来进行购买。

例子 9.34 测试 PayPal

```
var app = require('./../app');
var assert = require('assert');
var request = require('supertest');
var paypalMock = require('./paypalmock');
```

```
function makeCustomer() {❶
  return {
    address1: '123',
    city: 'Nottingham',
    country: 'GB',
    email: 'user@example.com',
    first_name: 'Paul',
    last_name: 'Smith',
    state: 'Nottinghamshire',
    zip: 'NG10932',
    tax_number: ''
  };
}
```

```
function makeOrder() {❷
  return {
    id: 1,
    customer: makeCustomer()
  };
}
```

```
function makePayPalIpn(order) {❸
  // More fields should be used for the real PayPal system
  return {
    'payment_status': 'Completed',
```

```
'receiver_email': order.customer.email,
'invoice': order.id
};
}

describe('buying the book', function() {
  var paypalServer;

  before(function(done) {
    paypalServer = paypalMock.listen(3001, done);
  });

  after(function(done) {
    paypalServer.close(done);
  });

  it('should redirect the user to paypal', function(done) {
    var order = makeOrder();

    request(app)
      .post('/buy')
      .send(order)
      .expect(302, done);

  });

  it('should handle IPN requests from PayPal', function(done) {
    var order = makeOrder();

    app.once('purchase:accepted', function(details) {
      assert.equal(details.receiver_email, order.customer.email);
    });

    request(app)
      .post('/paypal/success')
      .send(makePayPalIpn(order))
      .expect(200, done);

  });
});
```

- ❶ 固定的用户信息。
- ❷ 固定的订单信息。
- ❸ PayPal 会返回的信息。
- ❹ 每次测试开始前，开启 mock 的 PayPal 服务器。

⑤ 每次测试结束后，关闭 mock 的 PayPal 服务器。

⑥ 对于正确的订单，用户应该进行跳转。

⑦ 对于正确的订单，应该返回 200。

这个测试按顺序设置了一个样本②，首先需要一个消费者的信息①。我们还创建了一个对象，和 PayPal IPN 请求拥有同样的字段——也就是要发送到 PayPal mock 服务器进行验证的。在每个测试前④后⑤，我们还需要启动和关闭 mock 的 PayPal 服务。这是因为我们不希望运行不需要的服务，有可能导致其他测试有奇怪的行为。

当用户在我们网站上填写订单信息时，会 post 到一个路由，用于生成的 PayPal URL。PayPal URL 会将用户浏览器重定向到 PayPal 进行支付。例子 9.34 包括了这一点的测试⑥，它生成的 URL 是例子 9.32 中的本地测试的 PayPal URL。

这里还有一个关于 PayPal 发送通知消息的测试⑦。这是我们关注中需要 PayPal mock 服务器中的一点。首先，我们需要使用通知对象发起一个 POST 请求到服务器的 /paypal/success 地址③。这是 PayPal 通常会做的。然后我们的应用程序会发送一个 HTTP 请求到 PayPal，这时便会使用到 mock 服务器，返回 VERIFIED。这个测试只是简单地确保返回 200，但是它也能够用于监听 purchase:accepted 事件，这表示一个特定的购买行为完成了。

这看起来像是有大量的工作，一旦你的远程服务使用 mock 进行模拟后，便能够更加高效地工作。你的测试可以运行得更快，并且可以离线工作。也可以让 mock 服务器生成各种不同的响应，如果你想要，可以用来帮助获得更好的测试覆盖率。

这是本章内容中涉及的最后一个是 web 相关的技巧。下一节内容会讨论 Node web 开发的新趋势。

9.4 全栈框架

在这一章的内容中你已经看到了如何使用 Node 内置的模块、Connect 或者 Express 来构建 web 应用。有一种新的概念的框架，被称为全栈框架。这些框架提供的是创建基于浏览器和现在工具的富应用程序需要的特性，如数据绑定，同时也处理服务端所关心的业务逻辑层和数据持久化。

如果你打算使用 Express，也可以开始尝试一下使用全栈的框架。MEAN 是使用 MongoDB、Express、Angular 和 Node 技术栈的一种解决方案。有很多 MEAN 相关的实现，但 Linnovate 提供的一个（<https://github.com/linnovate/mean>）是目前最流行的。它使用 Mongoose 来处理数据层，Passport 来处理身份认证相关的，以及使用 Twitter 的 Boot-

strap 来作为用户 UI。如果你是在一个已经熟悉 Bootstrap、AngularJS 和 Mongoose 的团队，那么这是一个快速启动你们项目的方法。

*Getting MEAD*⁴ 这本书介绍了全栈开发相关的内容，包括 Mongoose 数据层、RESTful APIs 设计、Facebook 和 Twitter 的账户管理等。

另外一个基于 Express 和 MongoDB 构建的框架是 Derby (<http://derbyjs.com/>)。Derby 使用 Racer 替代 Mongoose 来实现数据层。使用 operational transformation (OT) 来允许来自多个客户端的数据可以是同步的。OT 被特别设计用于协同系统，所以 Derby 对于开发类似 Etherpad (<http://etherpad.org/>) 的应用是一个好选择。它提供了类似模板和数据绑定的客户端特性。

如果你喜欢 Express，但是想要更多的特性，可以考虑 Paypal 开发的 Kraken (<http://krakenjs.com/>)。这个框架在 Express 项目的基础上添加了更多结构，包括配置子目录和控制器，Grunt 任务和测试。它同时也提供了国际化的相关特性。

一些框架几乎都关注于浏览器端，而依赖于 Node 的只是比较敏感的操作和数据的持久化。一个比较流行的例子便是 Meteor (<https://www.meteor.com>)。和 Derby、MEAN 技术栈一样，它也使用 MongoDB，但是开发者们在计划添加其他数据库的支持。这个框架基于一个发布/订阅模式的架构，JSON 文档在客户端和服务端之间进行推送。客户端保持一个文档的复制——服务器发布文档集合的相关更新，客户端来进行订阅。这意味着在浏览器的大部分数据层相关的代码都是同步编写的。

Meteor 拥抱反射，一种范式，是目前在桌面开发领域相当流行的。这允许在方法中进行反射的计算的绑定。如果你对一个值进行了一个函数的订阅，当值改变时，函数应该会返回。这在一个真实的应用中最显著的影响便是流式的代码风格——这样会有更少的发布/订阅管理和事件绑定的代码。

Hoodie (<http://hood.ie>) 是 Meteor 的一个竞争对手。它使用 CouchDB，适用于移动端的应用，因为它尽可能地进行数据同步。几乎所有的操作都在本地完成。它包括了内置的账户管理，与 hoodie.account.signUp('alex@example.com', 'pass') 一样简单。有一个全局的公共存储，所以数据可以为特定的用户保存着，或者为每一个应用的用户开放。

还有很多活跃的 Node web 框架，但是仍有 Node web 开发的一个方面没有提及：实时的

⁴Simon Holmes 编写的书，*Getting MEAN*：<http://www.manning.com/sholmes/>。

应用开发。

9.5 实时服务

Node 天生就很适合做实时的 web 服务。从广义上讲,这涉及到三种类型的应用:统计服务器、协作服务,以及对延迟比较敏感的应用,如游戏服务器。

使用 Express 来创建一个服务器,用于收集你的其他应用,服务器、气象传感器、养狗机器人等数据,这并不困难。不幸的是,要处理好这个并不是很容易。如果当每次有人玩你的免费的 IOS 游戏时,你正在记录一些东西,每一分钟有成千上万个事件时会发生什么?你应该如何扩展,或者实时地看到关键的信息?

一些公司在巨大的规模下会有这样的问题,幸运的是他们其中一部分已经创建了我们可以使用的开源工具。其中一个例子是 Square 公司的 Cube (<http://square.github.io/cube>)。Cube 允许你收集带有时间戳的事件,然后获取它们的指标数据。它使用 MongoDB,所以你可以使用数据生成其他一些东西来创建图表。Square 有一个数据可视化的解决方案,称为 Cubism.js (<http://square.github.io/cubism/>),可以实时地进行更新值的渲染(见图 9.9)。

 Square

Cubism.js

Time Series Visualization

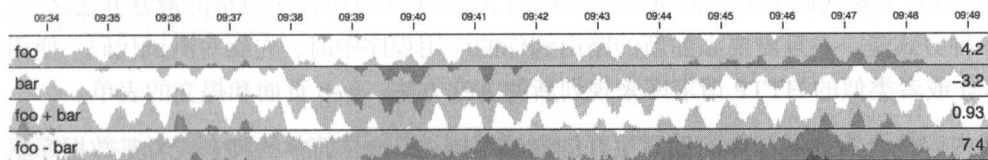


图 9.9 Cubism.js 实时展现时间序列的值

Etherpad (<http://etherpad.org/>) 项目是一个 Node 驱动文档协作编辑器。它允许用户在修改文档和更改代码颜色时可以交流,所以很容易看到每一个人在做些什么。它是基于你在这本书中已经见过的模块:Mikeal Rogers 的 request、Express 和 Socket.IO。

WebSockets 让这些项目变得可能。没有 WebSockets 的话,推送数据到客户端会变得比较

麻烦。Node 有很丰富的 WebSockets 相关的实现——Socket.IO (<http://socket.io>) 是最流行的，但也有 ws (<https://www.npmjs.org/package/ws>)，声称是最快的 WebSocket 的实现。

Sockets 和 streams 可以平行使用；SocketStream (<http://socketstream.org/>) 旨在成为填补构建完全基于 streams 的 web 应用空白的桥梁。它使用 HTML5 的 history.pushState API 来创建单页面应用，还有 Connect 中间件和浏览器的代码共享。

9.6 总结

在这一章中，你可以看到 Node 是如何在现代的 web 应用开发中发挥作用的。它可以用来改善前端工具——对现在前端开发人员来说，安装 Node 和 Node 构建工具是很常见的事情。

Node 也可以用于服务端的开发。Express 是比较流行的一个 Node 后端开发框架，但很多项目只是需要 Connect 的一个功能自己便可以了。其他类似于 Express 的框架，多少都有着不同的侧重点。例如 Restify，用于创建严格的 RESTful 应用 APIs。

编写结构良好的 Express 应用意味着你需要采用特定的一些模式或者 Node 社区推崇的一些良好实践方式。这里边包括错误处理（技巧 70），用文件夹区分模块和做路由分离（技巧 67），以及利用事件解耦（技巧 73）。

在浏览器中使用 Node 的模块（技巧 66）和在 Node 中使用客户端的代码（技巧 65）也会变得越来越常见。

如果你想要编写良好的代码，尽可能地采用测试驱动开发的方式。我们已经介绍了相关的技术，mock 远程服务的数据（技巧 78）来测试类似身份认证之类的程序（技巧 76），编写对于新代码的简单测试是提高你的 Node web 应用程序质量的最好办法之一。其中一种方式是，一旦需要添加新的路由到你的应用程序中时，先编写测试代码。使用 supertest 或者类似的 HTTP 请求库来规划新的 API 方法、web 页面和提交的表单。

下一章内容将会为你展示如何编写更好的测试用例，无论是简单的脚本或者是数据库驱动的 web 应用。

10

测试：编写健壮代码的关键

本章概要

- 断言、自定义断言和自动化测试
- 确保期望的异常被捕获
- Mocha 和万能测试协议（TAP）
- 测试 web 应用
- 持续集成
- 数据库测试装置

设想一下现在有这么一个需求，在一个在线商城里支持一个新的交易币种。为了实现这样一个需求，首先，你应该为该货币能够被预想到的一些相关计算定义一个测试，比如小计、计税、总计这样的计算。接着需要编写相关的实现代码来保证你所定义的测试能够成功通过。在这一章中，我们将会学习到如何使用 Node 内置的测试特性来编写测试。我们会使用 Node 内置的 `assert` 模块来测试你在 `package.json` 文件中设置的测试脚本，最后，我们也会介绍到两个主流的测试框架 Mocha 和 `node-tap`。

测试介绍

我们假设你在学习本章内容之前已经拥有编写单元测试的经验。但如果你没有相关的经验，那么请参考表 10.1，那里定义了将会用到的一些术语。如果你想知道什么是断言、测试用例或者测试装置，都可以在表格中找到相关的参考定义。

如果还想要了解更多关于测试的介绍，可以参阅 Roy Osherove 编写的《单元测试的艺术》第二版（*The Art of Unit Testing, Second Edition*），该书原版由 Manning 出版社发行（<http://manning.com/osherove2/>）。该书讲述了如何从零开始编写易维护、易读的测试代码。另外一本值得推荐的关于测试的书就是《测试驱动开发实战》（*Test Driven Development: By Example*），由 Kent Beck 和 Addison-Wesley 编著，2002 年发行（<http://mng.bz/UT12>）。

在很早之前，Node 社区就开始采用和应用测试了，这也是使用 Node 编写应用开发最大的优势。关于编写测试用到的 Node 模块也是应有尽有，这就使得你可以快速方便地编写出可读性很好的测试代码。你或许开始疑惑，为什么测试那么重要，为什么我们在开发之前就要编写测试呢？这么给你说吧，编写测试有助于你在真正实现一个需求或者是想法之前，加深和挖掘你对这个想法的理解——你可以把它们当成一些很小巧的实验。它们也帮助你更好地理解你将要做的事情，这也就是说，你编写的测试用例其实就是对需求描述的最好文档，它们能够扩展或延伸到项目开发中核心的需求。另外，编写好的测试用例也可以帮助你减少一个已经成熟的项目维护成本，比如，当你在对一个成熟的项目进行修改时，之前写好的测试就能够帮助你验证变更是否会导致现有的功能无法运行。

首先，我们要先了解 Node 的 `assert` 模块。这个模块允许我们定义一个对程序运行预期的结果，如果程序得到的结果和预期的结果不匹配，那么就会抛出异常。对结果的预期和描述就是测试的主要目的，所以会在这一章中看到很多很多的断言。尽管你不必非得使用 Node 的 `assert` 来编写测试用例，但实质上它和其他你可以能用的语言中的断言类库一样，Node 中的断言模块也是一个内置的核心模块，所以本章前面将会用大量的篇幅来介绍断言。

为了让每个人都能够快速读懂，我们将会在下面讲解一些常用的测试用语。

10.1 Node 测试的相关介绍

为了让新手学习自动化测试变得更简单，我们在下面的表 10.1 中的术语定义中包括了一些常见的术语，同时也罗列了一些将会用到的一些特定用语，因为不同的编程社区用到的术语可能还是略有不同的。

表 10.1 Node 测试的相关概念

术 语	描 述
断言	<p>一个运行你测试表达式的逻辑语句。由核心模块 <code>assert</code> 模块支持，比如下面的例子：</p> <pre>assert.equal(user.email, 'name@example.com');</pre>
Test case	<p>用一个或者多个断言来测试特殊的场景：</p> <pre>it('should calculate the square of a number', function() { assert.equal(square(4), 16); });</pre>
测试装置	<p>一个用于运行测试用例和整理测试运行结果的程序。该程序输出的结果可以用于诊断追踪测试失败的原因，下面这个例子基于前面的例子之上，展示了用 <code>Mocha</code> 作为测试装置的过程：</p> <pre>var assert = require('assert'); var square = require('./square');</pre> <pre>describe('Squaring numbers', function() { it('should calculate the square of a number', function() { assert.equal(square(4), 16); }); it('should return 0 for 0', function() { assert.equal(square(0), 0); }); });</pre>
夹具	<p>测试数据通常是在测试运行之前就准备好了的，比方说，如果你想测试一个用户账户系统，那么就需要预定义一些用户的账号和密码，然后在一些设计好的测试中判断使用相应的账号密码是否可以成功登录。</p> <p>在 <code>Node</code> 中，<code>JSON</code> 就是一种用于测试夹具中的文件格式，它非常流行。当然你也可以用一个数据库，或者是 <code>SQL</code> 的语句块，抑或是 <code>CSV</code> 文件来存储这些数据。这完全取决于你的程序的需求</p>

续表

术 语	描 述
模拟对象	一个用于模拟另外一个对象的对象。模拟对象通常用来替换一些 I/O 操作，通常这些操作要么是因为运行很慢或者是很难在单元测试里运行。比如类似于通过使用 WEB API 从远端下载数据，或者是访问一个数据库
桩	一个方法桩是用来替换测试中的一些功能的，这些功能用来占据测试的运行时间。比如说，模拟一个用于类似于磁盘这样的 I/O 源打交道的方法，或者模拟是一个可以从远端 API 返回数据的方法
持续集成服务器	一个用于运行自动化脚本的服务器，每当代码版本管理服务器的代码发生变化时，都会触发该服务器自动运行测试脚本

在上面表格中，Node 可以直接支持的特性就是断言。要支持其他特性，Node 就必须借助于第三方类库——我们将会在第 86 章介绍到持续集成服务器，在第 87 章中会介绍到模拟对象和测试夹具。当然，如果只是编写测试用例，你可能不会用到表格里所有东西，实际上，你完全可以只使用断言模块就编写出测试用例。下面，开始介绍断言模块，这样你就可以开始用它编写基础的测试用例。

10.2 使用断言编写简单的测试

至此，我们已经简要地提及到了，使用断言可以用来描述测试用例，但是这个过程需要其他什么东西吗？通常来说，断言就是一些用来判断是否和预期相匹配的函数，如果不匹配就抛出异常。打个比方说，一个产生失败结果的断言就如同你拿着一张无效的信用卡去一个商店里购物，那么可以断言，这样的情况下不管你刷多少次，都不可能刷卡成功。断言的概念其实很早就有了，甚至在 C 语言时代就已经有了断言的概念。

在 C 语言的标准库里就已经有了 `assert()` 的宏定义，它可以用来验证表达式。在 Node 里，我们也拥有一个核心的 `assert` 模块，当然尽管 Node 里有内置的 `assert` 模块，也还是有一些其他的第三方的断言模块的。

COMMONJS 单元测试

Node 中的 `assert` 模块是基于 CommonJS 的单元测试 1.1 实现的 (http://wiki.commonjs.org/wiki/Unit_Testing/1.1)。因此即使它是一个内置的核心模块，你也可以使用其他的断言模块，因为对应的相关概念其实都是大同小异的。

下面开始介绍内置的断言模块，接下来的第一个技巧，将会介绍使用 `assert` 核心模块中的 `assert.equal` 来检查相等性，并且将使用 `npm` 脚本来运行自动化测试脚本。¹

技巧 79 用内置的模块编写测试

你是否曾经有过这样的冲动：迫切地想为你的一个很重要的功能编写一个测试脚本，最后你却因为在阅读相关的测试类库的时候望而却步呢？的确，刚开始编写测试，总是很难的。然而如果使用 Node 的 `assert` 模块编写测试，那么压根就不需要使用其他特定的类库来编写测试脚本。

如果说你只是为一个很小的模块编写测试，那么通过一种可以省去安装各种依赖的方式来实现你的测试将会是很棒的。本技巧就会为你展示如何编写简洁而又富有表现力的单文件测试。

问题

你对类、模块或者方法的可以接受的输入输出有着很明确的定义和认识，当程序的输出结果和输入不匹配时，你想让程序有一个明确的反馈。

解决方案

使用 `assert` 模块和 `npm` 脚本。

讨论

断言模块在 Node 中是与生俱来的。你可以认为它就是一个专门用来验证输出和期望是否一致的一个工具。在内容的实现上，它是通过比较一个实际值 *actual* 和一个期望值 *expected* 来实现的。`assert.equal` 方法参数定义就很完美地展示了这一点，它们两个参数分别就是：`actual`、`expected`。当然，你也可以传入第三个参数 `message` 作为它的可选参数来代表一个消息，当测试结果失败的时候，人们也可以知道到底发生了什么事情。

现在我们举个例子，比如你现在编写一个这样的测试场景：你需要计算一个在线商城的订单价格，现在购物车里有 3 个商品，每件的单价都是 3.99 美元，你通过下面的测试来保证计算逻辑的正确性：

```
assert.equal(
  order.subtotal, 11.97,
  'The price of three items at $3.99 each'
);
```

¹这个是在 `package.json` 文件 `scripts` 属性中定义的，可以通过在命令窗口中输入 `npm help scripts` 来了解更多关于它的内容。

对于使用一个只拥有一个参数的 `assert` 方法，比如 `assert(value)`，期望的结果是 `true`，其模式和上面其实是一样的（补充说明：就是说，只要 `value` 不是 0 就返回真）。

可以尝试运行下面的代码，来发现测试失败的时候到底发生了什么：

例子 10.1 `assert` 模块

```
var assert = require('assert');  
var actual = square(2);  
var expected = 4;  
  
assert(actual, 'square() should have returned a value');  
assert.equal(  
  actual,  
  expected,  
  'square() did not calculate the correct value'  
);  
  
function square(number) {  
  return number * number + 1;  
}
```

- ❶ 加载断言模块。
- ❷ 断言模块就是用来验证传入的值是否为真。
- ❸ `assert.equal` 用来判断返回的值和设置的值是否相等。
- ❹ `square` 方法就是我们要测试的目标方法。

在很多的测试脚本中都会看到第一行代码，它用来加载 `assert` 模块❶。`assert` 变量也是 `assert.ok` 的一个别称——换句话说，你既可以使用 `assert()`，也可以使用 `assert.ok()`❷。

有时候很容易就忘记 `assert.equal` 的参数顺序，所以，你或许会发现为了这个还需要经常去翻阅 Node 的文档。然而其实这两个参数的顺序即使反了也没有什么关系——有些人可能会觉得把 `expected` 参数放到第一个会让他们在代码里查找对应的值变得更加简单——但是，不管怎么样，你最好保持前后定义的一致性。这也就是为什么在上面的例子中，我们显式地为变量命名成 `actual` 和 `expected`❸。

上面的测试用例中，需要测试的目标方法其实是意想得到错误的❹。所以当你通过运行上面的测试脚本 `node assertions.js`（上面的代码放在 `assertions.js` 文件中），你将会得到下面这样的有一个有错误提示的跟踪堆栈：

```
assert.js:92
  throw new assert.AssertionError({
    ^
AssertionError: square() did not calculate the correct value
    at Object.anonymous (listings/testing/assertions.js:7:8)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
```

❶

❶ 断言失败的文件和代码行数。

如果光从跟踪堆栈上来看，感觉很难发现到底错在哪里。但是由于上面的测试用例在断言方法中包含了一段消息，这段消息在错误发生的时候会显示出来，通过这消息的描述我们就可以发现怎么回事。当然，从堆栈跟踪的结果中，也可以发现它已经告诉我们错误发生在 `assertions.js` 的第 7 行❶。

除了上面的方法之外，在 `assert` 模块中还有其他用来验证测试结果的有用方法，其中有一个最重要的方法就是：`assert.deepEqual`，通过这个方法，我们可以判断两个对象的相等性，这个方法非常重要，因为 `assert.equal` 只能用来比较一些外围属性的相等性（补充：如果你了解浅拷贝和深拷贝的概念，对这个理解就会很明了）。浅等性用来比较一些原始属性或者字符串的相等，然而 `deepEqual` 可以更进一步地比较两个对象内置的对象和值。

当你在编写一些返回复杂类型对象的测试用例时，你也许会发现 `deepEqual` 很有用的。再回顾前面那个在线上商城的例子，你的购物车或许看上去是这样的：`{ items: [{ name: "Coffee beans", price: 4.95 }], subtotal: 4.95 }`，这个对象里包含了一组购物车对象和一个小计值，这个值是通过另外一个对象计算而来的。现在，如果要通过单元测试来验证这个对象，你就可以使用 `assert.deepEqual`，因为它可以用来比较对象而不仅仅限于原始类型的值。

下面的代码展示了 `deepEqual` 的使用：

例子 10.2 测试对象的相等性

```
var assert = require('assert');
var actual = login('Alex');
var expected = new User('Alex');
```

❶


```
assert.deepEqual(actual, expected, 'The user state was not correct');
```

❷

```
function User(name) {  
  this.name = name;  
  this.permissions = {  
    admin: false  
  };  
}
```

```
function login(name) {  
  var user = new User(name);  
  user.permissions.admin = true;  
  return user;  
}
```

❸

❶ 加载 assert 模块。

❷ 使用 assert.deepEqual 比较对象。

❸ 登录系统有一个错误。

这个例子中就用到了 assert 模块❶来验证一个从构造方法中返回到对象的值，例子描述的是一个虚构的登录系统。登录系统有时候甚至会错误地将一些管理员账户当成普通用户返回❸。

assert.deepEqual 方法❷就用于检查返回对象的每一个属性，验证是否和预期的对象的属性有所不同。当发现 user.permissions.admin 的值与期望比较的对象所对应的属性值不一样的时候，测试程序就会抛出 AssertionError 的异常错误。

如果查阅关于 assert 模块的文档，你还会发现其他一些很有用的方法，比如与之具有相反逻辑的方法 notDeepEqual 和 notEqual，甚至还有用于严格检查相等性的方法，类似于 === 的功能的 strictEqual 和 notStrictEqual 方法。

测试的场景往往不光仅限于上面讲述的验证值的相等性这样的场景，有时候我们也需要测试一些程序预期就应该发生的异常。接下来的技巧就会讲述如何测试程序的一些预期的错误和失败逻辑。

技巧 80 编写验证异常的测试

程序终究会有发生异常的时候，但是我们希望当程序发生异常的时候，能够输出一些有用的错误信息。这里要介绍的技巧就是如何保证程序抛出的异常是你预期发生的异常，以及会讲述到如何在测试过程中让程序抛出异常。

问题

你想编写一个测试用于测试代码的错误处理的逻辑。

解决方案

使用 `assert.throws` 和 `assert.ifError`。

讨论

Node 程序员必须知道的一个约定就是所有的异步方法必须将错误对象作为第一个参数返回。在设计我们自己的模块的时候，我们是知道什么地方可能发生异常的，理想状况下，需要保证这些错误被准确地返回到一些回调函数中。

下面的例子将会展示如何验证一个异常没有被预期地传到一个异步方法中。

例子 10.3 处理异步 API 中的异常

```
var assert = require('assert');
var fs = require('fs');

function readConfigFile(cb) {
  fs.readFile('config.cfg', function(err, data) {
    if (err && err.code === 'ENOENT') {
      cb(null, { database: 'psql://localhost/test' });
    } else if (err) {
      cb(err);
    } else {
      // Do important configuration stuff
      cb(null, data);
    }
  });
}

// 这个测试用于确保不存在的配置文件能够正确地处理
readConfigFile(function(err, data) {
  assert.ifError(err);
});
```

❶

❷

❸

❹

- ❶ 我们要测试的目标方法接受一个回调函数作为输入参数。
- ❷ 如果发生的异常“找不到文件”，则返回默认的值。
- ❸ 否则把错误传给回调函数。
- ❹ 这里如果有异常对象传入，`ifError` 就会失败。

尽管 `assert.ifError` 是同步运行的，但是它也可以在传递错误给回调的异步方法中使用来进行测试。例子 10.3 使用了一个异步方法：`readConfigFile` ① 来读取一个配置文件。在实际的程序使用过程中，这可能是一个 web 程序的数据库的配置文件，或者其他类似的配置文件；如果这个文件没有被找到，那么它就会返回默认值②。在这个方法运行的过程中，任何可能发生的错误都将被传入到一个回调方法中去③。

也就是说，用 `assert.ifError` ④ 可以很容易地检测是否有一些不是程序期望发生的异常。如果项目结构中的一些变更导致了一个不平常的异常错误发生了，这样一个测试就可以将这个异常捕获到，并且可以在程序发布之前警告开发人员程序中存在潜在的危险代码。

注意，在测试用例中处理抛出的异常的时候，我们没有使用 `try` 和 `catch`，而是使用 `assert.throws`。

为了使用 `assert.throws`，你必须提供一个运行的方法和一个异常构造器。因为当整个方法被传入的时候，它可以将异步的 API 很好地衔接，所以用它来测试一些依赖 I/O 操作的测试场景。

下面的代码将会展示如何在一个虚构的用户账户系统中使用 `assert.throws`。

例子 10.4 确保有抛出预期的异常

```
var assert = require('assert');
var util = require('util');
```

```
assert.throws(
  function() {
    loginAdmin('Alex');
  },
  PermissionError,
  'A PermissionError was expected'
);
```

```
function PermissionError() {
  Error.call(this, arguments);
}
util.inherits(PermissionError, Error);
```

```
function User(name) {
  this.name = name;
  this.permissions = {
    admin: false
```

①

②

③

```
};  
}  
  
function loginAdmin(name) {  
  var user = new User(name);  
  if (!user.permissions.admin) {  
    throw new PermissionError('You are not an administrator');  
  }  
  return user;  
}
```

❶ 传入的第一个参数就是需要被测试的目标函数。

❷ 第二个参数是期望抛出的异常。

❸ `PermissionError` 继承自标准的 `Error` 构造器。

❹ 这是一个模拟的登录系统，只允许管理员登录。

断言❶用来检查和确保期望的异常有被抛出。第一个参数就是我们需要测试的目标函数，在这例子中，就是 `loginAdmin` 函数，第二个参数就是期望的异常❷。

这里，我们需要强调两点关于 `assert.throws` 的事情：它可以被用作一个异步 API，因为你可以将函数方法传入，并且它期望函数抛出的是指定的某种类型的异常。在使用 Node 开发项目的时候，我们建议使用 `util.inherits` 从 `Error` 构造器继承得到一个错误类。这使得你可以很容易地捕获抛出的异常，同时还使你可以用一些额外的属性来修饰这些异常，比如当你需要增加一些有用的信息的时候。

在这个例子里，我们创建了 `PermissionError`❸，当人们在跟踪堆栈里看到这样一个异常抛出的时候，顾名思义，很容易知道发生了什么事情。接着，一个 `PermissionError` 的错误会在 `loginAdmin` 函数中抛出❹。

这个技巧使用的是 `assert` 模块来处理异常，结合上个技巧的内容，通过使用 `assert.equal`，你可以很快比较数字和字符串。这就已经可以覆盖测试很多我们在有关账务处理相关的 web 应用程序中发生的场景了，比如说，检查发票的金额、邮箱地址等。很多时候，`assert.ok` 被用作 `assert()` 的一个别称，使用它足够快速地验证一些真值表达式。然而如果要充分利用 `assert` 模块，接下来还有一个你需要掌握的知识，那就是如何创建自定义的断言。

技巧 81 创建自定义的断言

通过扩展 Node 内置的断言可以支持一些程序特定的表达式。有时候，发现一直在重复使用相同的代码来编写测试用例，但是你其实又希望有一种更好的方式。比如说，如果你想在 `assert.ok` 中通过使用正则表达式验证合法的有限地址。为了解决这样一个问题，你可以通过编写自定义的断言来实现。通过学习如何编写自定义的断言，也可以帮助你从内到外更好地理解断言模块。

问题

如果你拥有一个正确的断言，你可以替换掉测试代码中很多重复的代码。

解决方案

扩展内置的 `assert` 模块。

讨论

`assert` 模块建立在一个单独函数之上：`fail`，`assert.ok` 实际上就是调用的 `fail` 方法，传入一些取反的逻辑。它看上去就是这样的代码：`if (!value) fail(value)`，如果你剖析一下 `fail` 函数是怎么工作的，你就会发现它只不过是抛出一个 `assert.AssertionError` 的异常。

```
function fail(actual, expected, message, operator, stackStartFunction) {  
  throw new assert.AssertionError({  
    message: message,  
    actual: actual,  
    expected: expected,  
    operator: operator,  
    stackStartFunction: stackStartFunction  
  });  
}
```

这个错误对象里包含一些相关的属性，使得测试报告能够分析出错误发生的位置和原因。编写这模块的人知道其他人想编写他们自定义的断言，所以，`fail` 函数就暴露出来，这样的话，这个函数就可以被复用了。

编写一个自定义的断言需要包括下面这些步骤：

1. 定义一个与现有的断言类库里已有方法签名相似的方法。
2. 当与期望不匹配的时候调用 `fail` 函数。
3. 让测试产生一个 `AssertionError` 的异常。

在例子 10.5 中，我们综合了这些步骤定制了一个断言来确保一个正则表达式能够匹配。

例子 10.5 一个定制的断言

```
var assert = require('assert');❶  
assert.match = match;  
  
function match(actual, regex, message) {  
  if (!actual.match(regex)) {❷  
    assert.fail(actual, regex, message, 'match', assert.match);  
  }  
}  
  
assert.match('{ name: "Alex" }', /Alex/, 'The name should be "Alex"');❸  
  
assert.throws(❹  
  function() {  
    assert.match('{ name: "Alex" }', /xlex/, 'This should fail');  
  },  
  assert.AssertionError,  
  'A non-matching regex should throw an AssertionError'  
);
```

- ❶ 加载 assert 模块。
- ❷ String.prototype.match 调用一个正则表达式来验证一个字符串。
- ❸ 确保测试可以通过。
- ❹ 确保测试失败。

这个例子中加载了断言模块❶，然后定义了一个 match 方法，当传入的实际值和正则表达式所要求的匹配不上的时候❷，该函数里面通过运行 assert.fail 来产生期望的异常。这里你必须牢记的关键细节就是：定义的参数列表必须和断言模块里的其他方法保持一致——比如这个例子里定义的方法就是基于 assert.equal 的形式的。

例子 10.5 中也包含了一些测试，在实际的程序调用中，这个场景可能发生在一个单独的文件中，这里阐述了定制的断言是如何工作的，首先需要检查是否传入一个简单的测试通过用一个正则表达式来验证字符串❸，并且保证当测试失败的时候能够通过 assert.throw 抛出 AssertionError 异常❹。

你自己的业务语言

使用自定义的断言只不过是一种用于创建你自己的业务语言的技术。如果你发现你的测试用例中有很多的重复代码，那么这个时候说明你可以将这些重复的代码用一个类或者是方法来包装。

比如说，`setUpUserAccount({ email: 'user@example.com' })` 就比一个三到四行的设置代码要更易读，特别是当这样的代码在你的测试用例中重复出现时，这个就尤为有用。

这个例子看上去可能很简单，但是理解如何编写自定义的断言可以增强你对断言模块的理解。当你可以很容易地扩展内置的断言类库的时候，自定义断言可以帮助你清理测试。比如通过自定义断言实现这样的调用 `assert.httpStatusOK`，通过上面的技术，这个完全可以做到。

关于断言的讲解就到这里吧，下面一起来学习下如何组织不同文件里的测试。在下面的技巧中我们将会介绍通过测试装置来组织一组测试文件，这使得我们可以更加容易地执行这些测试。

10.3 测试装置

测试装置或者说测试框架，通常是一个用于设置运行环境并且用于执行测试脚本的程序，它可以收集和比较测试的结果，因为它是自动化的，所以测试脚本可以通过在持续集成的服务器上运行。关于持续集成的相关内容将会在技巧 86 中讲到。

测试装置可以用来执行一组测试文件，也就是说可以利用它很容易地使用一个命令来运行很多的测试脚本，这不仅使得执行脚本变得简单方便，同时也为你和别人协同工作提供了便利。你甚至可以决定在做任何事情之前选择好一个测试装置。接下来的技巧就会介绍如何制作属于你自己的测试装置，并且了解通过在 `package.json` 文件里如何添加脚本本来为你节省时间。

技巧 82 使用一个测试装置组织测试

设想你正在开发这样的项目，这个项目慢慢会变得越来越庞大，所以如果在项目中仅使用一个单独测试文件，会感觉到一片混乱。这将会导致代码的可读性变得越来越差，最后必然导致出错。所以你或许会想为某些相关的代码编写单独的测试文件。当程序遇到问题的时候，也许你只想每次执行一个文件来帮助你查找问题。

测试装置会解决这个问题。

问题

你想用测试用例和测试套件来组织编写测试。

解决方案

使用测试装置。

讨论

首先，让我们讨论什么是测试装置？在 Node 里，测试装置就是一个可运行的命令行脚本，在这个脚本中，你可以输入脚本名称来运行脚本。基本来说，它必须可以执行一组测试文件，并且当发生错误时，它能够将错误展示出来。当一个断言失败时，它必将导致一个异常抛出，对此我们不需要做任何特别的操作，要么该命令脚本将以返回编码 0 安静地退出执行。

也就是说，一个最基本的测试装置不过就是：`node test/*.js`，命令中的 `test/` 是一个包含了一组测试文件的目录。当然，我们可以对这样一个命令进行更好的处理，所有的 Node 程序都拥有一个 `package.json` 的文件，在这个文件中有一个 `scripts` 的属性，并且其中一个默认的脚本就是 `test`，这里你可以设置任意的字符串，就像在 shell 命令行里设置一样。

下面的代码就展示了 `package.json` 里设置的一个 `test` 脚本。

例子 10.6 一个关于 `test` 测试脚本的 `package.json` 配置文件

```
{
  "name": "testrunner",
  "version": "0.0.0",
  "description": "A test runner",
  "main": "test-runner.js",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "test": "node test-runner.js test.js test2.js"
  },
  "author": "",
  "license": "MIT"
}
```

❶

❶ 测试脚本在这里调用。

通过在上面的配置文件里使用 `node test-runner.js test.js test2.js` 将其设置为 `test` 的脚本^❶，这就使得其他的开发人员可以直接在命令行里输入 `npm test` 来运行你的测试脚本。这比单独记住一些特定的命令要方便简单得多了。

接下来，展开上面这个例子以进一步说明测试装置到底是怎么工作的。一个测试装置就是一个用于运行一组测试脚本的 Node 程序。因此，我们可以为这样一个程序提供一组需要测试的文件，一旦有任何一个测试失败，它都可以将相应的堆栈跟踪信息展示出来以方便我们分析失败的源头。

另外，每当有任何一个测试失败时，这个测试装置程序都会返回一个非零状态并退出运行。这就使得我们的测试可以以一种自动化的方式来运行，即其他的软件并不需要通过将实际的测试文本输出结果进行转换来判断测试失败与否，也就说这使得这个过程变得更加简单。这其实也就是持续集成服务器工作的原理：每当有新的代码往类似于 Git 这样的源代码管理器提交时，持续集成服务器就可以自动化运行所有的测试。

下边的例子展示了这个系统的一个测试文件看起来应该是怎么样的。

例子 10.7 一个测试文件实例

```
var assert = require('assert');
```

```
it('should run a test', function() {  
  assert('a' === 'a');  
});
```

❶

```
it('should allow a test to fail', function() {  
  assert(true);  
  assert.equal('a', 'b', 'Bad test');  
});
```

❷

```
it('should run a test after the failed test', function() {  
  assert(true);  
});
```

❸

❶ `it()` 函数声明一个测试用例。

❷ 我们可以看看当一个测试失败的时候将会是什么样的情况。

❸ 这个测试还会继续运行。

上面的 `it` 函数^❶看上去有点奇怪，但是它是一个全局的函数，这个函数是由测试框架提供的。它用来为每一个测试用例提供一个方便理解的名字。上面的例子中也包含了一个

会失败的测试场景^❶，这样我们可以看看当发生错误的时候测试装置会有什么反应。尽管前面的测试已经失败了，最后一个测试用例^❷会一直运行到结尾。

现在，我们来看看最后一个测试代码：接下来的代码清单里包含了一个可以用于执行例子 10.7 中测试文件的一个程序。

例子 10.8 使用指定的方式运行测试

```
var assert = require('assert');
var exitCode = 0;
var filenames = process.argv.slice(2);

it = function(name, test) {❶
  var err;

  try {❷
    test();
  } catch (e) {
    err = e;
  }

  console.log(' - it', name, err ? '[FAIL]' : '[OK]');❸

  if (err) {❹
    console.error(err);
    console.error(err.stack);
    exitCode = 1;
  }
};

filenames.forEach(function(filename) {❺
  console.log(filename);
  require('./' + filename);
});

process.on('exit', function() {❻
  process.exit(exitCode);
});
```

❶ 将 it 函数定义成一个全局的函数。

❷ 将测试以一个回调的方式传入到一个 try/catch 块中。

❸ 根据出现的异常打印出结果。

❹ 输出错误堆栈以有助于分析错误源头。

- ⑤ 每一个传入命令行的文件都会被运行。
- ⑥ 有测试失败，当程序退出的时候，返回一个非零的错误代码。

为了运行上面的例子，你可以将测试文件作为参数传入：`node test-runner.js test.js test2.js test-n.js`。上面的 `it` 函数定义成了一个全局的函数①，将其命名成 `it`，也使得整个测试和输出看上去符合人们的逻辑习惯。即使当有错误打印出来的时候③，看上去也是情理之中的事情。

因为 `it` 函数接收一个测试用例和一个回调，这个回调函数可以在我们所期望的条件下运行。在上面的例子中，我们在一个 `try/catch` 块中运行这个回调②，也就是说，可以通过 `catch` 的方式来捕获失败的断言并且将错误反馈给用户④。

上面的程序通过调用 `require` 将所有要测试的文件都以参数的形式传入到命令行中⑤。当然，如果要让我们上面的程序变得更加完善，那么文件处理的部分就需要变得更加复杂，就比如说，需要支持通配符。

当有测试用例失败的时候，`exitCode` 变量就会被设置上一个非零的值。并且这个值会在进程控制的事件 `process.exit` 中返回出去⑥。

尽管上面例子很小巧，我们也是可以通过使用 `npm test` 来运行它的，它通过 `it` 为测试用例提供一点点语法糖，为这个简单而又富含断言的测试文件的测试错误报告提供了更好的可读性，并且，当执行过程中遇到问题的时候，它也会返回一个非零的状态码。这一点，对于大多数流行的 Node 测试框架来说是最基本的，Mocha 就是这样的—一个测试框架，我们将会在下文的章节里介绍它。

10.4 测试框架

如果你正打算开发一个新的项目，那么我们建议首先还是尽早选择一个测试框架。打个比方说，如果你正在构建一个在线的博文系统，抑或是一个内容管理系统，那么你的系统必然会有这样的业务场景：允许一些人登录这个系统，但是只允许某些特定的用户能够看到管理员界面。通过使用像 Mocha 或者是 TAP 这样的测试框架，可以轻松构建出一些测试用例来解决上面特定的业务场景：普通用户的账户登录；管理员用户登录到管理员界面。你可以为这些特定场景建立各自的测试文件，或者将一组测试共同绑定在“用户账户测试”场景之下。

通过使用测试框架，你可以运行你的测试脚本，并且通过使用测试框架当中的一些特性，也使得编写和维护测试脚本变得更加简单。技巧 84 就会讲述 Mocha 测试框架，在技巧 85 中，我们会介绍万能测试协议 TAP (Test Anything Protocol) <http://testanything.org/>,

这两个测试框架目前是 Node 社区里最流行的两个测试框架。Mocha 是一个轻量级的测试框架，它可以用来运行脚本，提供三种用于组织脚本的方式，²这个框架期望你使用的是 Node 内置的 `assert` 模块或者是其他第三方的模块。相反，`node-tap` 实现了 TAP 协议，它自带的 API 包含了断言的功能。

技巧 83 使用 Mocha 编写测试

在 Node 中有很多测试框架供选择，选择一个正确的测试框架是一件难事。Mocha 这个框架是一个不错的选择，因为它是当前一个很流行的框架，目前也得到了很好的维护，并且它的功能也相对很全面，使用起来很方便。

通常来说，使用一个测试框架主要是用来组织你项目当中的测试。你肯定喜欢使用一个大家都熟知的框架，这样的话大家在合作的时候就不需要重新学习一个新的模块了。也许你也只是想寻找这样一种方式：可以每次都用相同的方式来运行脚本，或者是通过一个自动化的系统来触发运行测试。

问题

你需要使用一种其他开发人员都熟知的方式来组织你的测试脚本，并且通过这种方式，可以使用一个单独的命令来运行脚本。

解决方案

使用一个为 Node 服务的开源的测试框架，比如说 Mocha。

讨论

在此之前，你必先使用 `npm install` 安装 Mocha。安装 Mocha 最好的方式就是使用这样的命令：`npm install --save-dev mocha`。`--save-dev` 是选项参数，这样的话，Mocha 在安装到 `node_modules/` 目录的同时并且更新项目的 `package.json` 文件，在更新的时候，它会使用 Mocha 最新的版本，并且会将相关的信息保存到该配置文件的开发依赖当中。

Mocha 的版本

本章使用 1.13.x 版本的 Mocha。我们更倾向使用安装在项目本地目录下的 Node 模块来运行我们的脚本，而不是使用系统级别的 Node 模块来运行。也就是说，在调用 Mocha 的时候，使用的是 `./node_modules/mocha/bin/mocha test/*.js`，而不

²Mocha 支持使用 BDD（行为开发驱动）、TDD（测试开发驱动）和 Node 的模块系统的 API 风格。

是直接输入 `mocha` 命令。这就使得不同的项目可以用不同版本的 `mocha`, 以防我们的不同版本和主要发布的版本之间的 API 发生了很大的变化。

一个选择是使用 `npm install --global mocha` 来全局地安装 Mocha, 然后输入 `mocha` 来给一个项目运行测试。如果它找不到任何测试用例时会输出错误信息。

例子 10.9 展示了一个使用 Mocha 编写的简单测试。它使用到了 `assert` 的核心模块来做断言, 这些断言会在 `mocha` 的命令行类库中被调用。你应该把 `./node_modules/mocha/bin/mocha test/*.js` 加到 `package.js` 的 `"test"` 属性值当中去——详细操作请参考技巧 82。

例子 10.9 一个简单的 Mocha 测试

```
var index = require('../index');
var assert = require('assert');

describe('Amazing mathematical operations', function() {
  it('should square numbers', function() {
    assert.equal(index.square(4), 16);
  });

  it('should run a callback after a delay', function(done) {
    index.randomTimeout(function() {
      assert(true);
      done();
    });
  });
});
```

- ❶ 用 `describe` 将不同的测试组合在一起。
- ❷ 在一个异步的测试中包含一个 `done` 的参数。
- ❸ 当异步的测试结束时, 调用 `done`。

`describe` 和 `it` 函数都是由 `mocha` 提供的。`describe` 函数可以用于将一些相关的测试组织在一起。`it` 函数包含了来自一个测试用例的一个断言集合❶。

对于异步的测试, 我们还需要做一些特殊的处理。这包括需要为测试用例提供一个 `done` 作为回调方法❷, 然后在测试结束的时候调用这个回调❸。在上面的这个例子里, 在一个随机的间隔之后, 将会有有一个超时被触发, 也就是说, 我们需要在 `index.randomTimeout` 方法中调用 `done` 方法。下面的代码展示了需要在上面的例子中用于测试的测试代码:

例子 10.10 Mocha 测试例子

```
module.exports.square = function(a) {  
  return a * a;  
};  
  
module.exports.randomTimeout = function(cb) {  
  setTimeout(cb, Math.random() * 500);  
};
```

❶

❷

- ❶ 一个简单的同步方法，用于求平方值。
- ❷ 一个异步方法，会在一段随机的间隔之后被运行。

控制同步和异步行为

如果在 `it` 的参数中没有包含一个名为 `done` 的回调函数。那么 Mocha 将会以同步的方式来运行测试。在 Mocha 的内部实现中，它会检测你传入 `it` 的回调函数的参数中是否包含 `done` 作为回调函数参数。这一点是它用于判断测试是以同步还是异步来运行的一个开关。如果你传入这样的一个参数，那么 Mocha 会一直等到超时之后 `done` 被执行。

上面的例子中的模块定义了两个方法：一个是用于求数的平方值❶，另外一个则是通过在一个随机的超时之后调用一个回调方法❷。它们足够用于展示例子 10.9 中 Mocha 的特性了。

为了给你的项目设置 Mocha，在我们的例子里用到的 `index.js` 文件应该要放在项目自己的目录里，和它在同一个目录的文件应该是 `package.json` 配置文件，在该配置文件的内容中，包含一个 `scripts` 的属性，可以为这个属性的一个子属性 `test` 设置一个值：`"/node_modules/mocha/bin/mocha test/*.js"`，这里的 `/test` 目录应该包含了 `example_test.js` 文件。³当这些所有的东西都准备就绪了，就可以通过运行 `npm test` 来执行测试脚本了。

当测试运行起来了的时候，你会发现命令窗口中会出现一些点点（像省略号一样的点），这就标志着一个测试用例已经运行完了。当测试运行的时间超出了预计的时间，窗口上的字体颜色会有变化来告诉你测试运行慢到无法接受了。因为上面的例子使用的是 `index.randomTimeout`，可能因为随机取到一个非常大的值，这就使得你的测试需要运行很长的时间，这个时候 Mocha 就会认为测试运行得太慢了。当然，可以通过增加一个

³这个文件可以在 `test/` 目录下的任意地方被调用。

Mocha 的相关选项 `--slow` 来让那些运行得很慢的测试看上去不那么让人揪心。比如你可以这样做：`./node_modules/mocha/bin/mocha --slow 2000 test/*.js`。

每个测试都有断言

在例子 10.9 的代码中，每一个测试用例都只有一个单独的断言，有些人认为这种方式属于最佳实践——因为它可以使得你的测试可读性更好。

但是我们更喜欢每一个测试都只关注一个业务场景。这种方式让每一个测试用例的用意变得很明确，当然，这里也必须使用一定数量的断言。通常来说，这个数量可能不多，但是一般来说也会超过一个。

如果你想知道 Mocha 能够接受的所有选项参数，可以在你的命令行里输入 `node_modules/mocha/bin/mocha --help` 或者访问 <http://mochajs.org/> 了解相关知识。

最后，为了防止在编写 `package.json` 配置文件的时候遇到困难，我们将上面实例的 `package.json` 的代码提供给你。你可以通过执行 `npm install` 来安装 Mocha 和它的相关依赖。

例子 10.11 一个简单的 Mocha 测试项目的 `package.json` 配置文件

```
{
  "name": "mocha-example-1",
  "version": "0.0.0",
  "description": "A basic Mocha example",
  "main": "index.js",
  "dependencies": {},
  "devDependencies": {
    "mocha": "~1.13.0"
  },
  "scripts": {
    "test": "./node_modules/mocha/bin/mocha --slow 2000 test/*.js"
  },
  "author": "Alex R. Young",
  "license": "MIT"
}
```

在这个技巧中，我们使用到了 `assert` 的核心模块，但是如果你喜欢，可以使用其他第三方的 `assert` 类库的。类似的一些断言类库有 `chai` (<https://npmjs.org/package/chai>) 和 `should.js` (<https://github.com/visionmedia/should.js>)。

Mocha 常常用来测试 web 应用程序，在下面的技巧中，我们将会介绍如何使用 Mocha 来测试使用 Node 编写的 web 应用程序。

技巧 84 使用 Mocha 测试 web 应用

假设你正使用 Node 构建一个 web 应用程序，想通过一种可以发送请求和接受响应的方式来测试 web 程序——你想使用 http 请求来测试 web 程序是否按照期望的方式运行。

问题

你正使用 Node 构建一个 web 应用程序，并且想通过使用 Mocha 来测试它。

解决方案

使用 Mocha 并配合 Node 的 http 模块来编写测试。考虑设计一个 HTTP 的模块来简化测试代码。

讨论

理解 Node 的 web 应用程序测试的诀窍就是学会用 HTTP 来思考问题。本章的技巧就从一个 Mocha 的测试开始，并且配合 Node 的 http 的核心模块来展开。一旦掌握和理解了这种编写测试方法的原理，后面就会通过引入第三方的 HTTP 测试模块来展示如何简化测试。我们将会首先展示内置的 http 模块，因为这有助于理解背后到底发生了什么，然后在此基础上，学会如何构建这样的测试。

下面的代码展示了一个测试用例。

例子 10.12 一个测试 web 应用程序的 Mocha 测试

```
var assert = require('assert');
var http = require('http');
var index = require('../index');
```

```
function request(method, url, cb) {  
  http.request({  
    hostname: 'localhost',  
    port: 8000,  
    path: url,  
    method: method  
  }, function(res) {  
    res.body = '';  
    res.on('data', function(chunk) {  
      res.body += chunk;  
    });  
    res.on('end', function() {  
      cb(res);  
    });  
  });  
}
```

①

②

③


```
}).end();
}

describe('Example web app', function() {
  it('should square numbers', function(done) {
    request('GET', '/square/4', function(res) {
      assert.equal(res.statusCode, 200);
      assert.equal(res.body, '16');
      done();
    });
  });

  it('should return a 500 for invalid square requests', function(done) {
    request('GET', '/square', function(res) {
      assert.equal(res.statusCode, 500);
      done();
    });
  });
});
```

- ❶ 这个函数用于在测试中发送 HTTP 请求。
- ❷ 收到请求后，收集要发送给客户端的数据。
- ❸ 当请求和响应都结束后，执行回调函数。
- ❹ 确保响应是我们预期使用 `/square` 方法后的结果。
- ❺ 当遇到非法请求时候，可以抛出异常。

这个例子用于测试一个可以计算平方值的 web 服务。它是一个很简单的 web 服务，用于期望得到一个 GET 请求，并且能够以纯文本的形式给予响应。这个测试目的就是确保返回的值是自己期望的值，并且当遇到非法请求的时候，服务可以抛出异常。这个测试目的是在模拟浏览器的行为——或者其他的 HTTP 客户端，或者诸如此类的业务。服务端和客户端都运行在同一个进程中。

为了运行这个 web 服务，我们需要使用 `http.createServer()` 创建一个 web 服务器。具体的操作可以在下面例子 10.13 的代码中看到，在讨论例子 10.13 代码之前，让我们先看上面的例子。

上面的例子，首先创建一个实现 HTTP 请求的函数方法❶。这就节省了很多重复代码，不然类似代码将会在测试用例中到处都是。当然这个方法可以放到一个独立的模块中，这样就可以在其他的测试文件中被调用。当一个请求发出后，并且有数据从服务器上返

回时❷，它将会监听一个 data 事件。然后接着运行它提供的那个回调❸，这个回调是从测试用例中传入进去的。

图 10.1 展示了 Node 在开发 Web 应用程序的测试时如何让客户端和服务端在同一个进程中运行的。

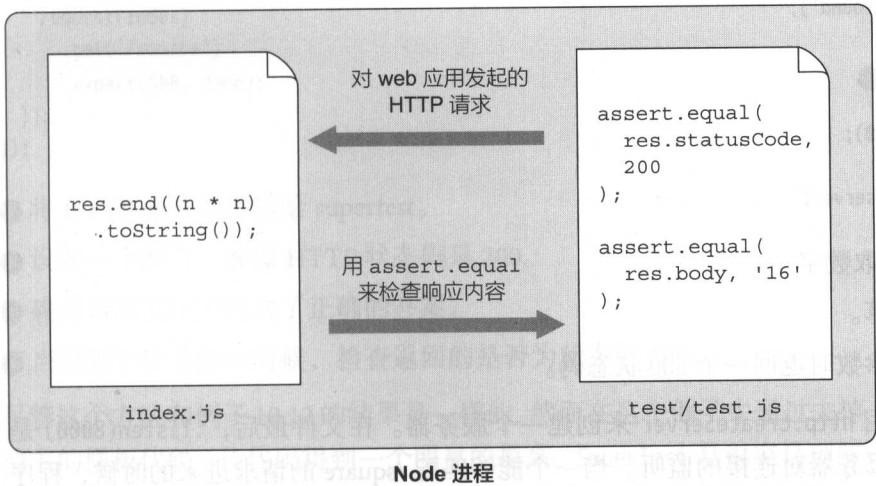


图 10.1 为了支持 web 应用程序测试，Node 可以在一个进程中同时运行一个 web 服务器和一个客户端请求

验证这个测试的一个用例就是使用 /square 来确保 $4 * 4 === 16$ ❹。一旦运行结束，我们也可以保证非法的请求会得到一个状态码为 500 的错误❺。

这是一个普遍使用的断言：判断返回的 res.statusCode 和预期的代码是否一致。

下面的代码展示了和上面例子对应的 web 服务的实现。

例子 10.13 一个可以求数的平方的 web 应用程序

```
var http = require('http');

var server = http.createServer(function(req, res) {
  if (req.url.match(/^\/square/)) {
    var params = req.url.split('/');
    var number;
    if (params.length > 1 && params[2]) {
      number = parseInt(params[2], 10);
      res.writeHead(200);
      res.end((number * number).toString());
    }
  }
});
```

❶

❷

```

    } else {
      res.writeHead(500);
      res.end('Invalid input');
    }
  } else {
    res.writeHead(404);
    res.end('Not found');
  }
})

```

```
server.listen(8000);
```

```
module.exports = server;
```

❶ 从 URL 中提取数字。

❷ 执行平方运算。

❸ 当遇到非法参数时返回一个 500 状态码。

首先，需要使用 `http.createServer` 来创建一个服务器。在文件最后，`.listen(8000)` 是用来让创建的服务器对连接的监听。当一个能够匹配 `/square` 的请求进来的时候，程序会从 URL 中提取出数字❶，然后再对数字进行求平方运算，再将结果返回发送给客户端❷。当期望的参数不存在的时候，程序就会返回一个 500 的错误状态码❸。

例子 10.12 中的代码有些地方是可以完善的，针对其中的 `request` 方法，我们可以使用现有的一些类库来代替清单中使用 `http.request` 来实现一个包装的代码。

我们可以选择 `SuperTest` (<https://github.com/visionmedia/supertest>)，这个模块由 TJ Holowaychuk 编写，此人参与过 `Mocha` 的开发。除此之外，当然还有其他实现类似功能的类库。它们都是用来简化 HTTP 的请求的，并且允许围绕发送的请求使用断言。

可以通过运行 `npm install --save-dev supertest` 将 `SuperTest` 添加到开发依赖中。

接下来的例子展示如何使用 `SuperTest` 来重构我们的代码。

例子 10.14 使用 `supertest` 来重构 `Mocha` 测试

```

var assert = require('assert');
var index = require('./../index');
var request = require('supertest');

describe('Example web app', function() {
  it('should square numbers', function(done) {
    request(index)

```

```
.get('/square/4')  
.expect(200)  
.expect(/16/, done);  
});  
  
it('should return a 500 for invalid square requests', function(done) {  
  request(index)  
    .get('/square')  
    .expect(500, done);  
});  
});
```

- ❶ 将 HTTP 服务器传送给 supertest。
- ❷ 设置一个断言，确保 HTTP 状态码是 200。
- ❸ 确保请求主体中包含了正确的答案。
- ❹ 当遇到非法参数的时候，检查返回的是否为状态码 500。

尽管这个方法和例子 10.12 的结果是一样的。然而在这个例子中通过去掉一些发送 HTTP 请求的样板代码，让代码得到一个明显的提高。SuperTest 是很容易理解的，也允许我们使用更少的代码申明断言，并且这些代码最终是通过异步方式运行的。SuperTest 暴露了一个 HTTP 服务器的实例❶。在这个例子中，它就是我们测试的那个应用程序。一旦这个应用程序被传到了 SuperTest 的主体函数 request 中，我们就可以使用 request().get 来发送 GET 请求。当然，它也支持其他的 HTTP 方法。譬如，可以将表单参数用 send 方法配合 post() 函数发送到服务器。

SuperTest 的方法是链式的，一旦你创建了一个 request，我们可以接着使用 expect 来设置一个断言。这是一个多态方法——它会依据检查传入的参数类型而执行不同的操作。如果传入的是一个数字❷，它将会用于确保验证 HTTP 的状态码是一个数字类型。如果使用一个正则表达式作为参数，它则用来验证一个响应字符串值是否与正则表达式匹配❸。这些都是我们在这个测试中完美期待的东西。

任何 HTTP 状态都会被检查到，所以，当我们期待得到一个 HTTP 500 的状态码的时候，上面的测试也是可以做到的❹。

尽管理解如何制作一个 web 应用程序，并且使用内置的 http 模块对它进行测试是十分有用的，但还是希望你能够了解如何使用第三方类库，比如 SuperTest，来让测试代码和测试用例变得更简洁。

Mocha 是当前 Node 中测试框架的主流框架，但是除了 Mocha 之外，还是有一些与之一样合格好用的框架的。下面技巧中，我们会介绍 tap 模块和和万能测试协议（TAP），因

为它是由 Node 的维护者和核心的开发者所认可和代言的。

技巧 85 万能测试协议 (TAP)

不同的程序语言和测试框架的测试装置的输出不尽相同，当前也有很多的尝试想在这些不同之上寻找统一。其中有个已经被 Node 社区采纳的尝试就是万能测试协议 (Test Anything Protocol) (<http://testanything.org>)，使用 TAP 产生的轻量级的流式结果可以被其他与之兼容的工具所使用。

假设你需要一个和 TAP 兼容的测试装置，因为你要么已经拥有使用 TAP 的其他工具，要么已经通过其他的语言已经熟悉它了。你或许并不喜欢类似 Mocha 的 API 方式，那么这个时候就可以用 TAP 来取代它，或许你也有兴趣去学习其他的测试解决方案，这些所有的假设，都会使得你了解和学习 TAP。

问题

你想使用一个设计用于和其他的系统进行交互的测试框架。

解决方案

使用 Isaac Z. Schlueter 的 tap 模块。

讨论

TAP 很独特，因为它的目标是通过制定一个协议，并且要求所有的测试框架都实现它，通过这样的一种方式将所有的测试框架和工具都联系起来。这里说到的协议是基于流的，它是轻量级的，并且具有人性化的可读性。相比其他的基于重量级的 XML 的标准，TAP 显得更加容易被实现和使用。

另外很重要的一点是，tap 模块 (<https://npmjs.org/package/tap>) 是有 Node 之前的维护者 Isaac Z. Schlueter 所编写的。这一点也充分印证了它在 Node 社区的高度影响力。

下面的例子我们还是继续用上面技巧 83 里的求数的平方值和有关随机超时的那个例子，这样也使得你可以将 TAP 和 Macha 有个对比。

下面的例子展示了 TAP 是如何工作的 (参考例子 10.10 与之对应的代码)。

例子 10.15 使用 TAP 编写测试

```
var index = require('../index');  
var test = require('tap').test;  
  
test("Alex's handy mathematics module", function(t) {
```

①

②

```
t.test('square', function(t) {  
  t.equal(index.square(4), 16);  
  t.end();  
});  
  
t.test('randomTimeout', function(t) {  
  t.plan(1);  
  index.randomTimeout(function() {  
    t.ok(true);  
  });  
});  
  
t.end();  
});
```

- ❶ 加载 TAP 模块，将其分配给 `test` 变量。
- ❷ 定义一个测试用例。
- ❸ 使用 `tap` 内置的断言功能。
- ❹ 调用 `end()` 表示测试已经运行结束。
- ❺ `plan()` 用来显示预期的断言数量。
- ❻ 当 `plan()` 被调用了，就没有必要再调用 `end()`。

这个例子和 Mocha 的例子有所不同的是，它并没有假设存在像 Mocha 里拥有的如 `it` 和 `describe` 这样的全局方法。首先，设置一个 `tap.test` 的引用❶。这个 `t.test()` 的方法可以用来定义测试用例❷，并且这种定义的方式是支持内嵌方式的。你可以根据需要用内嵌的方式来定义，内嵌的方式可以让你将相关的一些业务组合起来，所以在上面的例子中，我们也为每一个测试用例各自创建了一个方法。

`tap` 模块有内置的断言，如同我们在上面测试文件中使用的的那样❸。一旦你测试结束，必须调用 `t.end()` ❹，这是因为 `tap` 的模块假设所有的测试都是以异步的方式运行的，所以 `t.end()` 的方法可以在一个异步的回调中被调用。

另外一种方式就是使用 `t.plan` ❺。这个方法表示接下来预期有 `n` 个断言产生。一旦最后一个断言被调用，那么这个测试案例就算结束。和前一个的测试用例不同的是，第二个测试用例并不需要使用 `t.end()` ❻。

我们可以通过运行 `./node_modules/tap/bin/tap.js test/*_test.js` 来运行这个测试，当然，也可以通过在 `package.json` 文件里的属性 `scripts` 里设置 `test` 子属性的值，然后使用 `npm test` 来运行测试。

如果使用 `tap` 来运行测试，你看到的将会是很干净的输出，这些输出里面包含了每个断言的结果。这个是通过使用 `tap` 的一个子模块 `tap-results` 实现的。`tap-results` 模块的目的就是通过收集所有的 TAP 流，并且对跳过的、成功的、失败的测试结果分别进行计数，最后以一个简洁的报表形式输出。

```
ok test/index_test.js ..... 3/3
total ..... 3/3
```

ok

借助于 `tap` 模块的完美设计，你可以很方便地通过运行这样的命令来运行你的测试：`node test/index_test.js`。它将会打印出 TAP 的输出流：

```
# Alex's handy mathematics module
# square
ok 1 should be equal
# randomTimeout
ok 2 (unnamed assert)

1..2
# tests 2
# pass 2

# ok
```

使用 `tap` 模块编写测试，当测试运行失败的时候，它会返回一个非零的状态码并且退出执行。这个时候可以通过输入 `echo $?` 来知道返回的是什么状态码。你可以通过修改并运行上面例子 10.15 的代码来看看，如果失败之后输入 `$?` 得到的是什么结果。

TAP 通过设计产生和使用流的事实很符合 Node 本身的设计理念。在现实的开发过程中事实也是这样的，大部分开发项目中的测试都必须能够和一个自动化的系统进行交互。不管它是一个部署系统还是一个持续集成服务器。使用这种 TAP 协议，比使用重量级的 XML 标准的方式更简单，希望这种方式也能够在以后越来越流行。

图 10.2 展示了如何使用 `node-tap` 的子模块来测试一个程序。通过在不同的模块之间控制从测试转移到程序，最后再通过报告的方式输出结果，结果包含了测试的运行结果，有助于帮助你进行分析。必须认识到的一个关键点就是，`node-tap` 的子模块是可以被复用和替换的。如果你不喜欢 `tap-result` 的输出展现方式，可以用其他类似的类库替换它。

除了测试框架以外，真实的开发场景测试需要依赖更多重要的技术和工具。在下面一节中，我们会向你介绍如何使用持续集成服务器和数据库装置，以及如何模拟 I/O。

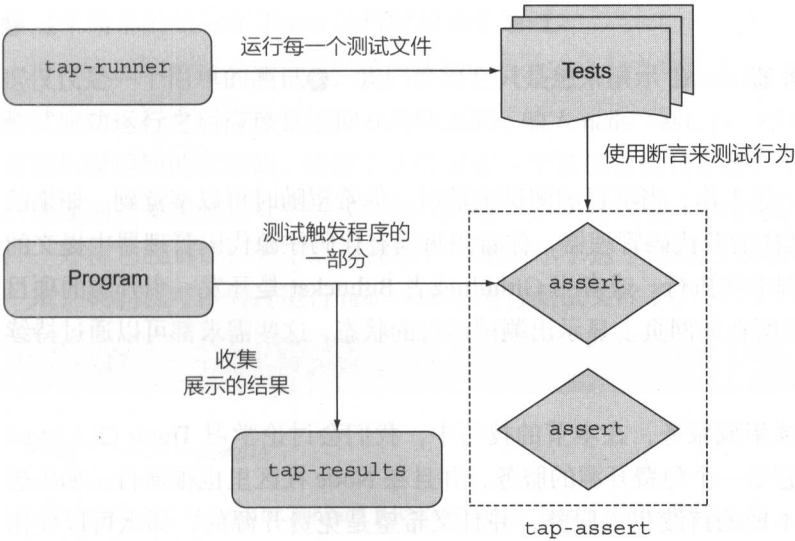


图 10.2 node-tap 使用了多个可复用的子模块来编排测试

10.5 测试工具

当你和一个团队一起工作的时候，因为有人提交了代码变更时导致已有的测试失败了，你会很想马上知道到底发生了什么。为了解决这样一个问题，我们在这节中会介绍如何设置持续集成服务器，在这一节中也会包含一些其他的技巧，用于讲解如何在测试中使用数据以及如何来模拟 web 服务。

技巧 86 持续集成

试想当你的测试都已经运行起来了，这个时候有人提交了一个会导致项目出现问题的代码，这个时候会发生什么呢？持续集成服务器用于自动运行测试脚本。因为所有的测试装置在测试失败的时候都会返回一个非零状态，从概念上来说它们都是足够简单的。如果你的测试装置配合了一些类似于 GitHub 这样的服务，当测试失败的时候，这个服务会向开发成员发送即时消息或者是邮件，这就使得结果变得非常清晰透明。

问题

当你的团队其他成员提交了会导致项目出问题的代码时，你希望能够察觉到这件事情，不至于发布出有问题的项目。

解决方案

使用一台持续集成服务器。

讨论

如果你是和一个团队一起工作，当项目的测试失败时，你希望随时可以察觉到。如果已经使用了类似于 Git 这样的源代码管理器，你希望每当有代码往源代码管理器中提交的时候就可以触发测试脚本被执行。或者当 Github 或者 Bitbucket 是开发一个开源的项目的时候，希望在代码库所在的网页上显示出测试运行的状态，这些需求都可以通过持续集成来解决。

现在有很多开源的持续集成服务，在本节的技巧中，我们会讨论学习 Travis CI (<https://travis-ci.org/>)，因为它是一个免费开源的服务，并且在 Node 社区里也很流行。如果想使用一个可以安装在本地的持续集成服务，并且又希望是免费开源的，那么可以使用 Jenkins (<http://jenkins-ci.org/>)。

Travis CI 提供了一张图片用于显示你的项目生成的状态结果。可以用 Github 的账号登录到 travis-ci.org，然后到 travis-ci.org/ 的 profile 页面，然后你在这个页面会看到你所有的放在 Github 托管的项目，只要通过打开一个那个显示状态图片的开关就可以让项目和这项持续集成的服务绑定在一起，这样的话，只要你的 Github 上的项目有任何更新，都会通知到 Travis CI 服务器。

如果你完成了刚才说的步骤，那么现在要做的就是往代码库里增加一个 `.travis.yml` 文件，这个文件用于告诉 Travis CI 你的项目是基于什么环境运行的。具体要做的只是设置好项目所用的 Node 版本。

下面我们通过一个完整的例子来构建一个使用 Travis 做 CI 的项目，通过这个项目就会知道 CI 是怎么工作的了。在这个例子中，你只需要 3 个文件，`package.json`、一个用于测试的文件以及一个 `.travis.yml`。下面就是具体的代码：

例子 10.16 使用 Travis CI 的一个简单测试文件

```
var assert = require('assert');

function square(a) {
  return a * a;
}

assert.equal(square(4), 16);
```


❶ 这个简单的测试在 Travis 运行时应该会通过。

这仅仅是一个简单的测试❶，我们使用它只是想演示 Travis CI 是怎么工作的。上面的测试成功运行之后应该是返回 0 的状态码。输入 `node test.js`，然后再输入 `echo $?` 来检查退出时返回的状态码。将这个文件放在一个新创建的目录里，并且后面我们会在这个新创建的目录里初始化一个 Git 的代码库，但在这之前，我们创建一个 `package.json` 文件，并将它放到该目录下。下面的代码就展示了这个完整的 `package.json` 的内容，这样一来，我们就可以通过运行 `npm test` 来运行测试文件了。

例子 10.17 一个基本的 `package.json` 文件

```
{
  "name": "travis-example",
  "version": "0.0.0",
  "description": "A sample project for setting up Travis CI and Node.",
  "main": "test.js",
  "scripts": {
    "test": "node test.js"
  },
  "author": "Alex R. Young",
  "license": "MIT"
}
```

最后，还需要一个 `.travis.yml` 文件，在这个文件里你要做的东西并不多，只需要告诉 Travis CI 你使用的是 Node 作为运行环境。

例子 10.18 Travis CI 配置文件

```
language: node_js
node_js:
  - "0.10"
```

接下来，登录到 [GitHub.com](https://github.com)，然后单击 **New Repository** 创建一个新的公共代码库，让我们将这个代码库命名成 `travis-example`，好让别人知道这个只是用来做演示用的。根据 GitHub 的相关教程，你可以通过使用 `git init` 来初始化你的代码库，并且通过使用 `git add .` 和 `git commit -m 'Initial commit'` 将目录里的文件添加到代码库。然后再通过运行 `git remote add <url>` 为你本地的代码库配置一个远端的地址，这里的内容就是你刚才在 GitHub 上创建好代码库之后得到的代码库的地址。最后你就可以通过 `git push -u origin master` 的方式将本地的代码库推送到远端的 Github 上。

到你的用户管理的 travis-ci.org/profile 页面把新项目的开关打开。你可能需要告知 Travis CI 来同步你的项目——单击靠近页面顶部的按钮。

在 Travis CI 上看到所有运行的测试之前，还有最后一个步骤，那就是你可以对 `test.js` 做些变更——再多添加一个你喜欢的断言。然后再使用 `git push` 将代码变更提交到远端。这个操作就会使得 GitHub 向 Travis CI 发送一个 API 请求，这个请求就会使测试被运行起来。

Travis CI 知道如何运行你的测试文件，它默认使用的是 `npm test` 运行脚本。如果在现有的项目里运用这项技术，也可以使用其他的命令（比如说 `make test`），这就需要修改 Travis CI 的 YML 文件。如果想了解更多关于如何配置项目生成的知识，可以参考这个文档（<http://about.travis-ci.org/docs/user/build-configuration/#script>）。

如果到你的 Travis CI 主页上，你可以看到一个日志控制窗口，上面有测试运行的具体细节信息。如图 10.3 所示就展示了当所有的测试运行成功了之后的情形。

Build	1	Commit	6238d35 (master)
State	Passed	Compare	ba59c09d5f54...6238d350b1c4
Finished	less than a minute ago	Author	Alex Young
Duration	16 sec	Committer	Alex Young
Message	Another test		


```
1 Using worker: worker-linux-7-2.bb.travis-ci.org:travis-linux-18
2
3 $ git clone --depth=50 --branch=master git://github.com/alexyoung/travis-example.git alexyoung/travis-example
4
5 $ cd alexyoung/travis-example
6
7 $ git checkout -qf 6238d350b1c4ff96ee73b47bf35dfc78c7746562
8
9 $ npm use 0.10
10
11 Now using node v0.10.12
12 $ node --version
13 v0.10.12
14 $ npm --version
15 1.2.32
16
17 $ npm install
18
19 $ npm test
20
21 npm WARN package.json travis-example@0.0.0 No repository field.
22 npm WARN package.json travis-example@0.0.0 No readme data.
23
24 > travis-example@0.0.0 test /home/travis/build/alexyoung/travis-example
25 > node test.js
26
27
28
29 The command "npm test" exited with 0.
30
31 Done. Your build exited with 0.
```

图 10.3 通过 Travis CI 运行脚本

既然你已经测试了测试脚本运行成功时候 Travis CI 展示的情形，那么也可以通过编辑 `test.js` 来测试当运行失败了又是什么情形。

在实际的开发项目中，可以为你需要运行的测试脚本在 Travis 中配置更多期望的东西，比如数据库和其他可以添加的服务（<http://about.travis-ci.org/docs/user/database-setup/>），甚至是虚拟机。

在项目中通过合适的装置配置数据库也是测试很重要的一部分。下面的技巧就会为你讲解如何为你的测试设置数据库。

技巧 87 数据库装置

大多数的程序都在某种程度上需要持久化数据。也就是说，对程序数据存储是否正确的测试显得尤为重要。在这个技巧中，我们会探索 Node 中处理数据库装置的方案，它们分别是：加载数据库转储、测试过程中创建数据、使用模拟技术。

问题

你需要在测试存储数据的代码，或者是执行一些类似的 I/O 操作，比如通过网络发送数据。你并不想在测试的过程中访问这个 I/O 源，或者想在测试之前预先加载你已经存在的测试数据。抑或是你的程序高度依赖 I/O 服务，你想仔细地通过代码来测试它们之间是怎么交互的。

解决方案

在测试之前预加载数据，或者模拟 I/O 层。

讨论

好的代码的标志是容易测试。用于执行 I/O 的代码感觉是很难进行测试的，然而如果代码的 API 完全解耦，其实也不难测试。

比如说，如果代码中存在执行 HTTP 请求的逻辑，就像前面的章节里讲到的技术一样，你可以通过在测试中定制化 HTTP 服务器来模拟一个远端的服务。这就是大家熟知的模拟对象技术。但是有时候可能不想模拟 I/O，你可能只是想编写可以直接让变化持久化到一个真正的数据库中的测试。尽管在测试中可以很安全地创建和销毁一个数据库实例。这样的测试就是大家熟知的集成测试，它们将软件不同的层集成起来，并且深度测试软件的行为。

在这个技巧里，我们会展示两种在集成测试中处理数据库装置的方式。接着，我们再拓宽话题，向大家讲解如何使用模拟技术。首先来看看通过数据库转储来预加载数据。

加载数据库转储

使用数据转储是数据库装置技术中的一个重头技术。所以你所需要做的就是运行所有的测试之前，腾出一个全新的数据库，并且通过将原始数据库拷贝到这个全新的数据库中。这样，所有的测试数据就从数据库中转储到了另外一数据库中。可以通过使用已有

的数据库工具将数据导入到新的数据库中，这样，你的测试就可以使用这个数据库里的数据。

在下面例子 10.9 的代码里，我们使用的是 Mocha 和 MySQL。当然，也可以使用其他数据库和测试框架，因为它们%的原理是一样的。如果想要了解更多关于 Mocha 的知识，回到技巧 83 中进行学习。

例子 10.19 assert 模块

```
var assert = require('assert');
var exec = require('child_process').exec;
var path = require('path');

var ran = 0;
var db = {
  config: {
    username: 'nodeinpractice',
    password: 'password'
  }
};

function loadFixture(sqlFile, cb) {
  sqlFile = path.resolve(sqlFile);
  var command = 'mysql -u ' + db.config.username + ' ';
  command += db.config.database + ' < ' + sqlFile;

  exec(command, function(err, stdout, stderr) {
    if (err) {
      console.error(stderr);
      throw err;
    } else {
      cb();
    }
  });
}

before(function(done) {
  ran++;
  assert.equal(1, ran);
  assert.equal(process.env.NODE_ENV, 'test', 'NODE_ENV is not test');
  loadFixture(__dirname + '/fixtures/file.sql', function() {
    process.nextTick(done);
  });
});
```

- ❶ 变量 `ran` 用于保证装置不会被多于一次加载。
- ❷ `loadFixture` 方法用于异步地准备数据库。
- ❸ MySQL 为导入数据转储做准备。
- ❹ 使用 `child_process.exec` 通过调用 `mysql` 的命令行工具来导入和复写数据。
- ❺ `before()` 回调会在所有其他的测试之前被调用。
- ❻ 使用断言确保数据仅被导入一次。
- ❼ 使用断言确保数据是在测试环境中被导入的。
- ❽ 执行数据导入。

这个例子中最基本的原则就是在其他运行执行之前导入数据。如果使用这种方法，请确保在这之前你的导入工具可以先将数据库清理干净。例如，如果在关系型数据库中，可以使用这样的语句：`DROP TABLE IF EXISTS`。

在运行上面的示例时，运行所有的测试文件之前，你需要把测试文件的名称传给 `mocha`。例如，如果在例子 10.19 调用的时候使用这样的代码：`test/init.js`，然后你就可以在 shell 命令窗口中运行这样的代码：`NODE_ENV=test ./node_modules/.bin/mocha test/init.js test/**/*.test.js`。或者通过在 `package.json` 文件中修改 `script` 节点的 `test` 属性的值。

上例中变量 `ran` ❶ 用于确保导入工具只运行一次 ❹，并且通过使用 `Mocha` 的 `before` 方法 ❺ 使得导入工具只运行一次，但是如果 `test/init.js` 意外地在其他地方被加载（可能通过 `mocha test/**/*.js`），那么导入就会发生两次。

为了导入数据，例子中定义了 `loadFixture` 函数 ❷，并且在 `before` 的回调中被调用 ❽。它可以接受一个文件名和一个回调函数作为它的参数，所以这个也就很容易以异步的方式运行。另外一个有必要的检查就是，确保这次执行是发生在测试场景中的 ❷。原因是其在基于 `NODE_ENV` 的应用程序也会去设置数据库，你并不希望数据库装置复写掉生产环境或者开发环境中所使用的数据库。

最后，这个通过导入数据的 shell 命令就构建好了 ❸，并且它是通过 `child_process` 来运行的 ❹。你可以使用其他任何的数据库——在例子里我们使用的是 MySQL 作为模板。同样的方法也适用于使用 MongoDB 或者是 PostgreSQL 或其他拥有命令行工具的数据库。

使用数据库转储技术有这样一些好处：可以使用你喜欢的数据库工具来维护测试数据（我们喜欢 Sequel Pro），它容易上手。如果更改了数据库架构或者是和数据相关的 `model` 类，那么你就需要更新数据库装置。

测试过程中创建数据

另外一种方式就是通过程序方式来创建数据。这种方式需要使用代码设置数据，这些代码通过在 `before` 回调里运行或者在其他测试框架中类似 `before` 方法的回调方法里运行，在这些代码里通过使用 `model` 类来创建数据库数据。

下面的例子就展示它到底是怎么工作的：

例子 10.20 使用 ORM 的方式来准备测试数据

```
var assert = require('assert');
var crypto = require('crypto');

function User(fields) {
  this.fields = fields;
}

User.prototype.save = function(cb) {
  process.nextTick(cb);
};

User.prototype.signIn = function(password) {
  var shasum = crypto.createHash('sha1');
  shasum.update(password);
  return shasum.digest('hex') === this.fields.hashed_password;
};

describe('user model', function() {
  describe('sign in', function() {
    var user = new User({
      email: 'alex@example.com',
      hashed_password: 'a94a8fe5ccb19ba61c4c0873d391e987982fbbd3'
    });

    before(function(done) {
      user.save(done);
    });

    it('should accept the correct password', function() {
      assert(user.signIn('test'));
    });

    it('should not accept the wrong password', function() {
      assert.equal(user.signIn('wrong'), false);
    });
  });
});
```



```
});  
});
```

- ❶ 一个用于替代模型数据的类。
- ❷ 模拟非阻塞式的数据存储。
- ❸ 为这个测试创建用户数据。
- ❹ 在测试开始之前保存用户。

这个使用 Mocha 运行，尽管它并没有使用一个真实数据库层，User 类❶模拟了你可能在一个关系型或者是非关系数据库的类库中看到的相关行为。上述例子里定义了一个 save 函数，并且拥有异步方式的 API ❷。所以这个测试就像在实际生活中发生的场景一样。

在 describe 块中，它将每一个测试用例进行分组，并且定义了一个 user 的变量❸。它将会在接下来的测试用例中被使用到。这个变量定义在它们的范围之上，所以所有的这些测试用例都可以访问到这个变量，但是也因为我们想在 before 块里以异步的方式来持久化数据。这个持久化的工作会在我们所有的测试之前就运行❹。

需要为测试数据而使用 ORM 吗？

像例子 10.19 的数据库转储技术一样，通过 ORM 来创建测试数据对于集成测试来说是很有用的，因为在这些集成测试中，你是真的想和数据库服务器打交道的。相比数据库转储技术，这需要更多的编程工作，但是如果你想通过调用在数据之上的 ORM 层的方法，ORM 就尤为有用。接下来的技巧就会通过在很多的文件里做更改来对数据库架构进行一定的修改。

使用模拟技术

最后要讨论的一个方法就是通过 API 模拟数据库的技术。尽管需要编写一些集成测试，但是也可以编写一些从来不需要触碰数据库的测试。相反，你可以将数据库的 API 抽象出来。

在 JavaScript 中，我们可以对已经定义好的对象进行修改。也就是说，你可以通过数据库模块来定义你自己的方法，并在这些方法中返回测试数据。现在已经有设计好了的类库可以让这个流程变得更加简单明了。其中在这方面表现的很好的一个就是 Sinon.JS。下面的例子将会通过使用 Sinon.JS 配合 Mocha 来为数据库模块打桩。

例子 10.21 的代码展示了一个为用户账户数据库使用 Redis 来打桩的例子。这个测试目的是来验证用户密码的加密功能是否运行正确。

例子 10.21 数据库打“桩”

```
var assert = require('assert');
var sinon = require('sinon');
var db = sinon.mock(require('../db'));
var User = require('../user');

describe('Users', function() {
  var fields = {
    name: 'Huxley',
    hashedPassword: 'a94a8fe5ccb19ba61c4c0873d391e987982fbbd3'
  };
  var user;

  before(function() {
    user = new User(1, fields);
    var stub = sinon
      .stub(user.db, 'hmget')
      .callsArgWith(2, null, JSON.stringify(fields));
  });

  it('should allow users to sign in', function(done) {
    user.signIn('test', function(err, signedIn) {
      assert(signedIn);
      done(err);
    });
  });

  it('should require the correct password', function(done) {
    user.signIn('wrong', function(err, signedIn) {
      assert(!signedIn);
      done(err);
    });
  });
});
```

- ❶ 模拟数据模块。
- ❷ 当用户登录的时候会使用这个哈希密码。
- ❸ 为 Redis 的 `hmget` 方法创建一个桩。
- ❹ 让 `hmget` 调用传入的回调，它是第三个参数（索引为 2），并且传一个 `null` 作为回调，接着是需要使用的字段。

这个例子是一个大型项目中的一部分，在这个项目中包含一个 `package.json` 文件和一个用于测试的 `User` 类——它在我们的示例代码中是可用的，在 `testing/mocha-sinon` 目录下。

在上面例子的第三行，你会看到一些之前没有看到过的代码：`sinon.mock` 包装了整个数据库模块^❶。这是一个我们已经定义过的数据库模块，它用于加载 `node-redis` 模块，然后连接到数据库。在这个测试中，并不想连接到一个真实的数据库，所以我们调用 `sinon.mock` 来模拟一个。这种方法可以被应用于其他使用 MySQL、PostgreSQL 或者其他数据库的项目中。只要你设计的项目是围绕着数据配置的，那么都可以使用模拟技术来包装它。

下面我们将为这个 `user` 设置一些字段^❷。在一个集成测试中，这样的字段总是从数据库里返回的。我们在这里并不想那么做，所以在 `before` 的这个回调函数里，使用了一个桩来重新定义 Redis 里 `hmget` 的功能^❸。这种定义桩的 API 方式是链式的，你可以在它的定义后面继续添加其他想做的事情，比如上面的代码里，我们紧接着调用了 `callsArgWith` 方法^❹。

`.callsArgWith` 方法的语言有些让人感觉迷惑，所以这里我们再讲解一下它到底是怎么工作的。在 `User` 类里，`hmget` 其实是像这样工作的：

```
this.db.hmget('user:' + this.id, 'fields', function(err, fields) {  
  this.fields = JSON.parse(fields);  
  cb(err, this);  
}.bind(this));
```

正如你上面看到的，它接收 3 个参数，记录的键值，你要取数据的哈希值，最后是一个可以拥有两个参数的回调函数，它们的两个参数分别作为可选参数的一个错误对象和要其加载到的数据。用这样的方式设置好桩之后，我们需要告诉 Sinon.JS 需要怎么样的响应。因此，`callsArgWith` 的第一个参数就是回调的索引，也就是 2，接着是这个回调可以接受的参数。我们将错误信息参数传入 `null`，然后用户的字段信息序列成字符形式。最后得到这样的形式：`callsArgWith(2, null, JSON.stringify(fields))`。

这个测试是很有用的，因为这个测试的意图是保证用户可以登录，并且仅限于使用正确密码的用户。这里用户登录的代码并不需要访问数据库，所以最好通过预定义好一些值而不是直接去数据库里取值，这样可以省掉不必要的麻烦。因为这里代码将 JSON 序列化到 Redis 中，我们不需要一个特殊的类库来序列化和编码 JSON——我们可以通过使用内置的 JSON 对象。

到现在，你应该已经明白什么时候使用集成测试、模拟和桩了。只要在正确的场合应用这些技术，它们都可以帮助你编写更好的测试。下面的表 10.2 对这些技术做了个总结，解释了什么情况下使用哪个技术。

表 10.2 什么时候使用持续集成测试、模拟、桩

技术名称	什么时候使用它
集成测试	<p>用于测试多组的模块。我们已经用它对使用访问真实数据库的测试和使用一种兼容性的 API 替换数据库访问的测试加以区别。在使用集成测试的时候，应该确保数据库是以你期待的方式运行的。</p> <p>集成测试有助于验证系统性能，但是它也高度依赖于测试数据，这就导致测试需要和数据库高度耦合，也就是说需要修改你的数据和数据操作的 API，也许还需要修改测试代码</p>
数据库转储	<p>这是一种需要在测试之前往测试数据库里预加载数据的方式。需要在准备数据之前做很多的工作，并且一旦数据库结构发生变化，这些测试数据也要需要及时维护。由于这种方法使用起来很简单，所以这多出来的数据维护工作也并不会是一个负担——你不需要使用特定的局来创建 SQL、Mongo 或者是其他的数据文件。</p> <p>在为一个已经拥有数据库的项目编写测试的时候，你可以使用这种技术。也许你刚从其他的编程语言或平台上转移到 Node 上，并且正在使用已有的数据库；你可以使用生产数据——但是你需要小心，不要泄露或者删除一些个人的信息，或者其他敏感信息——由此产生的数据库需要导入到你的项目代码库中</p>
ORM 装置	<p>宁愿在运行所有的测试文件之前创建一个导入文件，还不如使用 ORM 的方法在你的测试代码中创建并存储数据。时间长了，这个就很难维护——每一次数据库结构的变更，都需要小心地更新你的测试。</p> <p>当算法和当前的数据库里的数据有密切相关的时候，你可以选择使用这种方式。记住，让数据出现在你需要测试的代码的旁边，这样，即使出了问题，也容易查找和解决</p>
模拟对象和桩	<p>模拟对象是用于模拟其他对象的对象，在本章中，使用的是 Sinon.JS，这个类库可以用来处理模拟对象和桩。</p> <p>当你不想访问 I/O 资源的时候，可以考虑使用模拟技术。比如说，你正在编写一个需要和支付提供商类似 WorldPay 或者是 Stripe 这样的服务打交道的时候，那么可以通过创建一些类似于 Stripe 的模拟的 API，而不用实际和它们进行通信。通常来说，相比让你的测试文件直接访问 Internet，这种方式也更加安全，所以，所有需要使用网络的场景都可以使用模拟技术</p>

下次当你在编写测试远端 web 服务的测试或者是编写和数据库打交道的测试时，你应该知道怎么处理了。如果你觉得这一章的知识很有意思，那么可以去找更多的资料了解更多关于测试技术的知识。

10.6 扩展阅读

测试是一个很大的话题，尽管这一章已经讲了很多。但是还有很多重要的话题可以去考虑。Node 社区也一直在不断探索编写更好的测试方式，并且也慢慢将这种思想带到了客户端的开发中了，有一个类似的开发就是 Browserify (<http://browserify.org>)，它使得 Node 中的一些模块化模式和一些核心的模块，像 EventEmitter 和 stream.Readable，可以直接在浏览器中使用。

一些 Node 的开发人员用 Browserify 来开发更好的客户端测试。不仅充分利用了流和 Node 模块中更加清晰的依赖管理的模式，也可以在客户端使用 Mocha 或者是 TAP 这样在服务器端使用的技术。Browserify 的作者，James Halliday 创建了 Testling，它是一个用于通过浏览器在客户端运行自动化脚本的模块。

结合持续集成服务器，还有一些有用的和测试相关的用于检查代码的覆盖率工具，它们可以帮助我们分析当项目测试代码运行后对程序代码的覆盖情况，有助于构建更好的项目。这些报告可以显示出代码中的函数、方法甚至是 if 语句是否在你的测试中都被覆盖了，也就是说，如果在这些没有覆盖到的代码中可能就存在错误的隐患，就会导致你将有错误的代码发布到生产环境中。

10.7 总结

这一章中，学习了如何编写断言以及如何扩展它们，也学习了如何使用两个流行的测试框架。在 Node 项目中编写测试的时候，需要特别注意测试代码的可读性——你的测试应该做到快速，但是如果它们没有传达出测试的意图，那么在将来的维护上就会很麻烦。

下面来回顾一下本章学习的知识：

- 掌握了 assert 模块，学习了如何确保异常被正确地处理的各种方法。
- 使用类似 Mocha 和 TAP 的测试装置编写出更具可读性和维护性的测试代码。
- 通过使用一个加载数据的数据库或者是桩和模拟对象技术编写测试。
- 通过使用第三方类库，如 Sinon.JS，提高桩和模拟对象的编写。
- 为测试开发自己特定的领域语言——在编写方法和类的时候，保持测试用例简洁明了。

到现在，你还有一个没有涉及的知识点就是 Node 程序的调试，这在软件开发中是很重要的一个环节，当然，这取决于你的开发方式和背景。如果你对 Node 的调试基础知识很感兴趣，那么可以在接下来的章节中就会学习到在 Node 中调试代码的知识。

11 调试：用于发现和解决问题

本章概要

- 处理未捕获的异常
- 对 Node 应用进行语法检测
- 使用调试工具
- 分析应用程序和检查内存泄漏
- 使用 REPL 来检查程序执行
- 跟踪系统调用

在任何一个平台，构建稳定的应用程序，最重要的是理解错误是如何产生的，以及如何去处理。好的错误内省和内置的测试是调试问题最好的方式。在本章中，我们重点关注应该如何准备，以及如何防止应用程序变得糟糕。

当你的程序使用内存超过预期，或者 CPU 使用率高达 100%，很可能导致你的程序崩溃。在 Node 应用中，我们需要了解如何去调试来找出对应问题的解决方法。

第一部分内容我们会讲述 Node 应用对于错误处理和检测的相关设计。第二部分我们会尝试来调试特定的部分类型问题。

11.1 内省

设计应用程序时，我们需要考虑到如何去处理发生的错误，例如错误日志记录、错误干预。可以简单理解为，在哪里可能出现错误，那么我们需要捕获并且处理。在本书中，我们会介绍在 Node 应用中可能发生的各种形式的错误。在这里我们会囊括全部来进行讲解。

11.1.1 显式异常

那一些明确由 `throw` 关键词触发的异常抛出我们称为显式异常。它们即表示了有错误发生了：

```
function formatName (name) {  
  if (!name) throw new Error("name is required");  
  ...  
}
```

显式异常用 `try/catch` 块来进行处理：

```
try {  
  formatName();  
} catch (err) {  
  console.log(err.message, err.stack);  
}
```

如果你尝试抛出自定义的异常，那么有一些规则需要铭记于心：

- `throw` 只能用于同步方法中。或者当异步方法中异步执行触发前 `throw` 会生效（例如误用 API）。
- 通常抛出的异常对象都要继承于 `Error`。使用简单的字符串（如 `throw "Oh no!"`）无法获取对应的调试堆栈，即无法获取哪里发生错误的相关信息。
- 不要在内置的 Node 方法中的回调函数中抛出异常；这样子捕获到的堆栈没有什么有效的信息。我们可以直接处理异常或者把异常传递给另外的适合处理错误的函数。

又可以使用 `throw` 了

如果结构支持，你可以在异步块中也使用 `throw` 了。例如部分非稳定的特性，如 `domains`、`promise`、`generators`。

11.1.2 隐藏的异常

隐藏的异常是指在 JavaScript 运行时不是由 `throw` 关键字触发的异常。不幸的是，这些异常很容易就溜进我们的代码中。

其中一个很常见的异常是 `ReferenceError`，当变量或者属性的引用未找到时会发生。

这里，我们来看一个因为 `data` 的错误拼写导致的异常：

```
function (err, data) {  
  res.write(dat); // ReferenceError: dat is not defined  
}
```

另外一个很常见的异常是 `SyntaxError`，最出名的就是使用 `JSON.parse` 来解析无效 JSON 数据：

```
JSON.parse("undefined"); // SyntaxError: Unexpected token u
```

使用 `try/catch` 块把 `JSON.parse` 的使用给包裹起来是个好主意，特别是当你无法控制输入的 JSON 数据时。

尽早发现隐藏的异常

一种很棒的方式来发现隐藏的异常是使用类似 `JSHint` 或者 `JSLint` 的工具。把它们加入到你构建的过程中来保障代码得到有效的检查。这一章后续我们会了解更多关于这方面的内容。

11.1.3 错误事件

在 Node 中，错误事件可以在任意继承 `EventEmitter` 的对象中触发。如果没有相应的处理，Node 则会抛出错误异常。这些事件如果未绑定处理函数，是调试中最有难度的，因为可能在异步处理过程中会发生多次错误，类似有极少调用堆栈的 `stream` 数据：

```
var EventEmitter = require('events').EventEmitter;  
var ee = new EventEmitter();  
ee.emit('error', new Error('No handler to catch me'));
```

代码将会输入：

```
events.js:72  
  throw er; // Unhandled 'error' event  
  ^  
Error: No handler to catch me  
    at Object.<anonymous> (/debugging/domain/ee.js:5:18)  
    at Module._compile (module.js:456:26)
```

```
at Object.Module._extensions..js (module.js:474:10)
at Module.load (module.js:356:32)
at Function.Module._load (module.js:312:12)
at Function.Module.runMain (module.js:497:10)
at startup (node.js:119:16)
at node.js:902:3
```

幸运的是，我们知道这个错误来自哪里：就在我们刚写的那一点点代码里。但是在大型的应用中，有可能是因为 DNS 层出现错误，而我们无从得知哪个模块用了 DNS 而导致的问题。

所以，尽可能地绑定错误处理事件：

```
ee.on('error', function (err) {
  console.error(err.message, err.stack);
});
```

当编写自定义的 `EventEmitters` 对象时，在自己的执行代码以及定义的 API 中发生错误时，需要提供依赖的相关上下文，来抛给上层以便调试。同样地，使用 `Error` 对象而非字符串，以便找到对应的错误堆栈信息。

11.1.4 错误参数

在异步操作中发生的错误通常会作为回调函数的第一个参数传递过去。这和我们前边提到的错误不一样，这些通常不会直接引起异常。但是它们可能是部分隐藏的异常的源头。

```
fs.readFile('/myfile.txt', function (err, buf) {
  var data = buf.toString();
  ...
});
```

如上，我们忽略 `readFile` 返回的错误，假定获取的文件数据已经返回了对应的 `buffer`。不幸的是，当某一天无法读取相应的文件时，就会因为 `buf` 无法获取而抛出一个 `ReferenceError` 异常。

这是处理异常错误很棒的方法，这意味着很多时间，简单地把错误传入对应的错误处理函数即可优雅地处理好。

```
function handleError (err) {
  console.error('Failed:', err.message, err.stack);
}

fs.readFile('/myfile.txt', function (err, buf) {
  if (err) return handleError(err);
});
```



```
var data = buf.toString();  
...  
});
```

处理好我们提及的这四种类型的错误，可以让我们在遇到需要调试的问题时来获取更多有用的帮助信息。

尽管有很多优秀的实践经验和工具，但是我们还是会发现一些导致程序崩溃的异常。我们需要了解一下如何设计应用来确保可以快速定位和修复发生的异常。

技巧 88 处理未捕获的异常

如何有效处理 Node 程序的崩溃？首先，你需要明白当一个未捕获异常抛出时，Node 会终止进程的执行。重要的是理解为什么会存在这种行为，以及如何处理未捕获的异常，来帮助程序变得稳定。

问题

有未处理的异常导致程序无法正常运行。

解决方案

记录错误日志，并且优雅地处理掉。

讨论

有时候，程序会抛出未处理的异常，而 Node 默认会终止程序的执行。关于这种处理方式有一个很好的理由，我们过后再提。先来看一下如何修改这一默认的行为。

在 process 对象上可以设置未捕获异常的处理函数，当异常发生时，Node 会执行对应的函数，而不终止程序的运行。

```
process.on('uncaughtException', function (err) {  
  console.error(err);  
});
```

是的！这样子你的 Node 应用就不会崩溃了！尽管进程再也不会导致程序崩溃，但是如果你选择继续运行，该应用可能会泄露内存资源，并且可能变得极其不稳定，这样可能得不偿失。

这一切会是怎么发生的？我们看一下一个需要长时间运行的例子，一个 web 服务器。当有未捕获异常抛出时，我们不允许 Node 终止程序，而仅仅是打印错误信息。你觉得当一个用户请求时，程序有一个未捕获的异常时会有什么情况发生？


```
var http = require('http');

var server = http.createServer(req, res) {
  response.end('hello world');
});

server.listen(3000);

process.on('uncaughtException', function (err) {
  console.error(err);
});
```

❶

❶ 当 response 未定义时抛出一个引用异常。

当一个请求进来时，一个抛出的异常被 `uncaughtException` 处理函数所捕获。这个请求会怎么样？这会导致内存泄漏，请求会一直保持着，直至用户端请求超时（我们也没有办法去提供一个返回信息）。

在图 11.1 中，你可以看到泄漏发生的情况。如果没有异常，那么一切都是正常的。但如果异常发生了，那么便会出现内存泄漏。

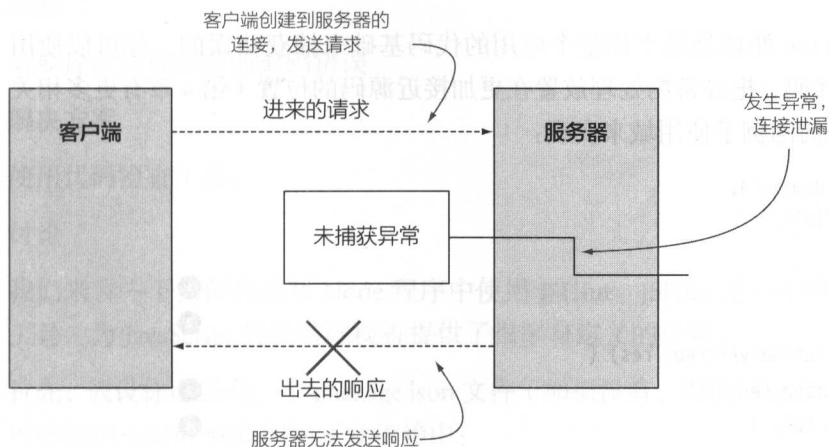


图 11.1 使用 `uncaughtException` 时的资源泄露

虽然这个例子被简化以便更加容易理解，但是未捕获的异常是一个现实的问题。大多数时候，它可能是打开的 `sockets` 或者文件无法正确地关闭。未捕获的异常通常在代码中埋得很深，这会让确定资源泄露的问题变得异常困难。

所在的背景情况也会受到影响，因为没捕获的异常会跳出当前的上下文，而让你身处于一个完全不同的上下文（`uncaughtException` 处理方法），让你丢失了错误相关的对象引用。在我们的例子中，我们没有办法访问到 `res` 对象来返回一个请求给客户端。

那么 `uncaughtException` 的好处是什么？它让你的应用可以适时打印日志以及重启。把 `uncaughtException` 处理方法作为对你即将崩溃的应用说再见的最后机会是很明智的做法。打印错误信息，发送一个邮件或者用其它方式通知相关人员，然后优雅地把应用关闭。

```
process.on('uncaughtException', function (err) {  
  console.error(err);  
  server.close();  
  setTimeout(process.exit, 5000, 1);  
});
```

❶
❷
❸

❶ 打印日志。

❷ 服务器停止接收请求。

❸ 等待 5 秒，让其他连接运行后结束进程。

`uncaughtException` 的处理函数是最后一道防线。理想情况下，异常应该更加接近发生时来进行处理，以防止泄漏和不稳定来源。对于这一点，你可以使用 `domain`。

使用域来处理未捕获异常

虽然 `uncaughtException` 处理是基于你整个应用的代码基础来捕获错误的，你可以使用域来控制监控部分代码，把异常的处理放置在更加接近源码的位置（第 4 章有更多相关内容）。让我们把先前的例子使用域来实现：

```
var domain = require('domain');  
var http = require('http');  
  
var d = domain.create();  
d.run(function () {  
  var server = http.createServer(req, res) {  
    d.on('error', function (er) {  
      res.statusCode = 500; )  
      res.end('internal server error');  
      server.close();  
      setTimeout(process.exit, 5000, 1);  
    })  
    response.end('hello world');  
  })  
  server.listen(3000);  
});
```

❶
❷
❸
❹

❶ 创建一个新的域。

❷ 在域中执行代码。

③ 在域中处理异常情况。

④ 给用户提供错误反馈。

使用域允许我们的代码在一个沙盒中运行，并且可以使用 `res` 对象来给用户反馈，这是上一个例子无法实现的点。尽管我们可以返回错误信息，并且关闭连接，但是最好的实践还是关闭进程。

如果你使用域，那么把各种各样的异常情况放到一个域的异常处理函数中处理也是个好主意，或者可以在本域中抛出异常而不会影响其他域。

另外我们来看一下一个有效的预防程序异常的方法：代码检查。

技巧 89 检查我们的 Node 代码

代码检查工具可以帮助我们适当地调整我们的代码。在前边的一些例子中，例如拼写错误导致的异常，虽然代码看起来是没问题的，但是访问未定义的变量，代码检查工具是可以查出这种错误的。

问题

需要提前发现潜在的编码错误。

解决方案

使用代码检查工具。

讨论

我们来看一下如何在编写 Node 程序中使用 JSHint。JSHint 是一个维护良好的代码检查工具，为 JavaScript 代码基础检查提供了很多自定义的选项。

首先，假设你已经有一个 `package.json` 文件（如果没有，则使用 `npm init` 来创建）。接着，可以添加 `jshint` 到我们的开发依赖中：

```
npm install jshint --save-dev
```

现在可以配置 JSHint 了。我们只需要在项目根路径放一个 `.jshintrc` 文件，用于编写 JSON 格式的配置。来看一下 Node 代码检查的基本配置：

```
{  
  "node": true,  
  "undef": true  
}
```

①

②

① 配置 JSHint 是用于检查 Node 代码，以防 Node 特有的全局变量使用时报错。

❷ 检查未定义变量。

JSHint 有很多的配置项（参考 <http://jshint.com/docs/options/>），可以配置来符合你的编码风格和意图，但是这些只是基础的一些配置。

需要执行 JSHint，可以在 package.json 中的 "scripts" 项添加以下代码（如果没有 "scripts" 这一项，可以自行补上）：

```
"scripts": {  
  "lint": "jshint *"  
}
```

❶

❶ 在项目中对所有的 JavaScript 文件进行 JSHint 检查。

接着你可以在项目目录下执行命令来运行 JSHint：

```
npm run lint
```

JSHint 将会打印出哪些地方出现错误，并且会提示可以通过修改代码或者更新配置来更好地符合代码风格和缩进。和测试一样，在提交代码前的自动构建过程中加入代码检查非常有用，有的时候会忘记去运行，如果能够自动化那是最好。

现在我们知道如何去有效预防异常的发生，以及如何在代码中处理异常。接下来看一下有哪些工具可以帮助我们在运行时去调试问题。

11.2 问题的调试

我们已经有了代码测试、日志记录和代码检查。但当问题发生时，我们应该如何去调试？幸运的是，在不同的场景都有对应的工具来帮忙。在本节中，我们将看看应用程序运行时，可能遇到的各种各样的问题，以及如何利用技术来解决这些问题。我们将开始使用调试器，然后尝试进行性能、内存泄漏等分析，进行生产调试以及代码跟踪。

技巧 90 使用 Node 内置的调试器

当你需要对应用程序进行一步一步的运行状态分析时，调试器便是最好的工具，Node 内置的调试器也不例外。Node 内置的调试器可以帮助你查看运行时变量，通过断点暂停代码执行，在部分应用代码出入，查看回溯，运行可以感知上下文环境的交互式 REPL，等等。

不幸的是，很多开发者都回避使用命令行工具，因为一看起来就令人生畏。我们要打破这一点，看一下调试器是多么强大，它提供的大部分功能已经足够帮你调试问题。

问题

你需要通过调试器来设置断点、查看变量，以及一步一步执行应用。

解决方案

使用 node debug 。

讨论

让我们以一个简单的程序，来试一下 Node 调试器的功能：

```
var a = 0;
function changeA () {
  a = 50;
}
function addToA (toAdd) {
  a += toAdd;
}
changeA();
addToA(25);
addToA(25);
```

要运行内置的调试器，使用 debug 命令即可：

```
node debug myprogram
```

应用便会以 debug 的模式开始，并且在第一行执行的代码中断运行：

```
< debugger listening on port 5858
connecting... ok
break in start.js:1
  1 var a = 0;
  2
  3 function changeA () {
debug>
```

查看所有可以使用的命令以及调试变量，可以使用 help ：

```
debug> help
Commands: run (r), cont (c), next (n), step (s), out (o),
backtrace (bt), setBreakpoint (sb), clearBreakpoint (cb),
watch, unwatch, watchers, repl, restart, kill, list, scripts,
breakOnException, breakpoints, version
```

从默认的断点开始，需要继续的话可以输入 cont ，或者用缩写的 c 替代。因为我们没有设置其他断点，应用会直接运行：

```
debug> cont
```

```
program terminated
debug>
```

当我们还停留在调试器中时，可以使用 `run` 命令来重新运行程序（缩写的 `r` 也可以）：

```
debug> run
< debugger listening on port 5858
connecting... ok
break in start.js:1
  1 var a = 0;
  2
  3 function changeA () {
debug>
```

回到调试器的内容中来，我们还可以在运行过程中使用 `restart` 命令来重新运行应用代码，如果需要，也可以使用 `kill` 来停止应用的执行。

这一段程序都和变量 `A` 有关系，所以我们使用 `watch` 表达式来监测程序运行中变量 `a` 的变化情况。`watch` 函数接受一个表达式参数作为监测的目标：

```
debug> watch('a')
```

我们使用 `watchers` 命令可以查看监测的所有内容的状态：

```
debug> watchers
  0: a = undefined
```

当前我们停在 `a` 赋值为 `0` 的代码执行前，所以 `a` 为 `undefined`。使用 `next` 命令（缩写为 `n`）来进入下一行代码的执行：

```
debug> next
break in start.js:11
Watchers:
  0: a = 0

  9 }
 10
 11 changeA();
 12 addToA(25);
 13 addToA(25);
debug>
```

这是相当方便的，进入下一步的代码运行时，调试器还可以打印出我们监测的内容，只需要再输入 `watchers`，我们便可以看到类似的结果：

```
debug> watchers
  0: a = 0
```

如果需要移除监测的表达式，使用 `unwatch` 命令，传入和创建时一样的表达式即可。

默认情况下，调试器只是打印当前上下文环境的前后两行代码，如果我们需要查看更多，可以使用 `list` 命令，传入需要查看的在当前停止的周围代码行数即可：

```
debug> list(5)
6
7 function addToA (toAdd) {
8   a += toAdd;
9 }
10
11 changeA();
12 addToA(25);
13 addToA(25);
14
15 }));
debug>
```

当前我们在第 11 行代码，即 `changeA` 函数。输入 `next`，会进入下一行代码的执行，即 `addToA` 函数。我们需要深入查看 `changeA` 函数时，可以使用 `step` 命令（缩写为 `s`）来进入函数的内部：

```
debug> step
break in start.js:4
Watchers:
0: a = 0

2
3 function changeA () {
4   a = 50;
5 }
6
debug>
```

现在我们在函数中，可以使用 `out` 命令在任何时间跳出函数内部。当执行到函数结束时，则会自动跳出，所以使用 `next` 也是可以的，我们试一下：

```
debug> next
break in start.js:5
Watchers:
0: a = 50

3 function changeA () {
4   a = 50;
5 }
6
```

```
7 function addToA (toAdd) {
debug>
```

可以看到，`watchers` 更新了输出，变量 `a` 现在的值为 50。紧接着是下一行：

```
debug> next
break in start.js:12
Watchers:
  0: a = 50
```

```
10
11 changeA();
12 addToA(25);
13 addToA(25);
14
debug>
```

又回到了 `changeA` 函数的下一行了。我们可以进入下一个函数。还记得是什么命令吗？

```
debug> step
break in start.js:8
Watchers:
  0: a = 50
```

```
6
7 function addToA (toAdd) {
8   a += toAdd;
9 }
10
debug>
```

我们来探讨一下调试器另外一个神奇的功能：内置的 REPL。我们可以通过 `repl` 命令来访问：

```
debug> repl
Press Ctrl + C to leave debug repl
>
```

这是一个标准的 REPL，包含了你使用 `repl` 时的上下文环境。举个例子，我们可以输出 `toAdd` 的参数：

```
> toAdd
25
```

还可以为应用程序引入其他状态，例如创建一个全局变量 `b`：

```
> b = 100100
```


这就是标准的 Node REPL，很多你可以在其他地方使用的功能，在这里也可以实现。

在任意情况下使用 Ctrl-C 便可以退出 REPL。我们尝试一下，现在你可以从调试的提示中看到，已经退出了 REPL 了。

```
debug>
```

在 REPL 中待了一会，好像忘记停留的上下文环境，可以再次使用 list 来查看一下：

```
debug> list()
3 function changeA () {
4   a = 50;
5 }
6
7 function addToA (toAdd) {
8   a += toAdd;
9 }
10
11 changeA();
12 addToA(25);
13 addToA(25);
```

是的，我们刚才停留在了第 8 行。正如你所知道的那样，我们想要 changeA 让 a 赋值为 100。这对于变量 a 来说是一个多么漂亮的数字。但是我们忘了，这还是在调试模式下。没问题，我们可以使用 setBreakpoint（缩写为 sb）设置断点来保留当前场景。

```
debug> setBreakpoint()
3 function changeA () {
4   a = 50;
5 }
6
7 function addToA (toAdd) {
*8   a += toAdd;
9 }
10
11 changeA();
12 addToA(25);
13 addToA(25);
debug>
```

请注意现在第 8 行代码前边多了一个星号（*）来表示我们在该位置设置了一个断点。在代码中修改当前函数并且保存：

```
function changeA () {
  a = 100;
}
```

回到调试器，重新运行应用：

```
debug> restart
program terminated<
debugger listening on port 5858
connecting... ok
Restoring breakpoint debug.js:8
break in start.js:1
  1 var a = 0;
  2
  3 function changeA () {
debug>
```

程序重新启动了，而设置的断点则还是原封不动。我们的修改有效了吗？看一下：

```
debug> list(20)
  1 var a = 0;
  2
  3 function changeA () {
  4   a = 100;
  5 }
  6
  7 function addToA (toAdd) {
  8   a += toAdd;
  9 }
 10
 11 changeA();
 12 addToA(25);
 13 addToA(25);
 14
 15 });
debug>
```

另外一种在程序代码里正确设置断点的方式是使用 `debugger` 关键字：

```
function changeA () {
  debugger;
  a = 100;
}
```

如果我们再次重新启动程序，会停留在 `debugger` 的代码行上边。我们可以使用 `clear-Breakpoint`（缩写为 `cb`）来清除设置的断点。

我们看另外一个主题，未捕获的异常。例如，在 `changeA` 函数中的 `ReferenceError` 异常：

```
function changeA () {
  a = 100;
  foo = bar;
}
```

使用 `restart` 重启应用，并且使用 `cont` 跳过一开始的断点，程序会因为一个未捕获异常而崩溃掉。我们可以使用 `breakOnException` 来跳过异常：

```
debug> breakOnException
debug>
```

现在，程序不会崩溃了，而是跳过异常，并且允许我们在程序终止运行前检查代码的状态。

有效的多文件调试命令

上述的场景只是关注于一个代码文件，而没有包括多个模块。调试器同样支持一些可以帮助你调试多个文件的命令。使用 `backtrace`（缩写为 `bt`）来获取当前停留位置的调用堆栈。你也可以使用 `scripts` 来获取当前所在的代码文件以及已经加载的文件。

如果你习惯使用 GUI 调试工具，那么内置的调试器可能会让你感觉不是很舒服。但是它实际上已经相当强大。只需要一个 `debugger` 语句便可以快速调试起来。

技巧 91 使用 Node Inspector

希望使用内置的调试器的所有功能，但是也想要有 Chrome 的开发者工具的界面？有一个模块就是干这个事情的！它叫 `node-inspector`。在这一部分内容中，我们会看一下如何使用它来进行程序的调试。

问题

需要利用 Chrome 开发者工具来调试 Node 应用。

解决方案

使用 `node-inspector`。

讨论

Node 暴露了一个调试端口，提供给第三方模块或者工具进行使用（包括内置的调试器），来进行远程的调试。其中一个很流行的模块是 `node-inspector`，它把调试信息从 Node 拿出来，并且放到 Chrome 开发者工具中。

开始使用 `node-inspector`，只需要安装即可：

```
npm install node-inspector -g
```

别忘记使用 `-g` 标识进行全局安装。安装好了之后，可以使用下边的命令来运行：

node-inspector

现在 node-inspector 已经运行，并且会告知你如何使用：

```
$ node-inspector
Node Inspector v0.7.0-2
  info - socket.io started
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

现在可以使用任意的 blink 内核的浏览器，如 Chrome 或者 Opera，来浏览这个 URL。但是我们现在没有 Node 程序可以进行调试，所以看到一个错误信息，如图 11.2 所示。

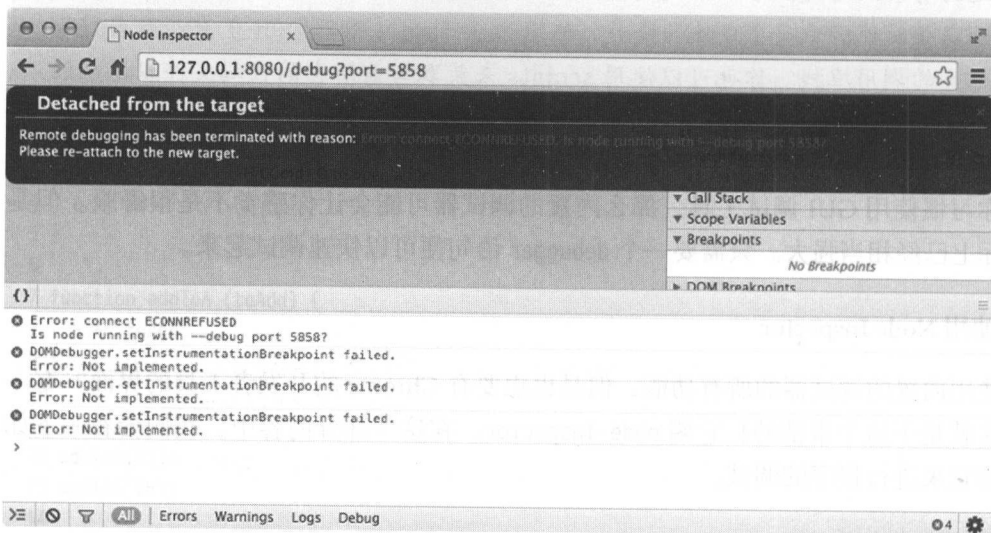


图 11.2 当找不到调试代理时的错误显示

先来编写小程序来试一下：

```
var http = require('http');

var server = http.createServer();
server.on('request', function (req, res) {
  res.end('Hello World');
});
server.listen(3000);
```

现在我们使用暴露的调试接口来运行这个程序：

```
$ node --debug test.js
debugger listening on port 5858
```

通过程序的提示可以看到，调试器监听端口 5858。如果我们刷新 node-inspector 的 web 页面，一切就很有趣了，如图 11.3 那样。

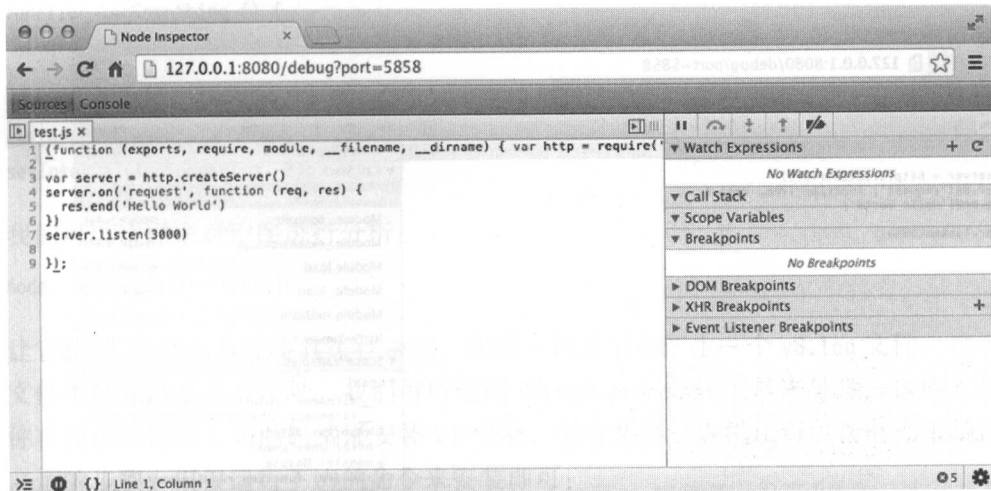


图 11.3 已经连接到调试器的 Node inspector

我们可以像内置的调试器一样，使用 inspector 来设置断点和监测变量。它同时也提供了一个控制台，可以像 REPL 一样，当应用程序暂停时，来浏览相应的状态。

node-inspector 和内置的调试器有一点不同的是，它不会自动在一开始的时候设置默认断点。如果需要，可以使用 --debug-brk 参数来开启：

```
node --debug-brk test.js
```

这个可以让调试器在第一行代码中暂停执行，以便进行步进调试或者控制继续执行。我们重新加载 inspector，可以看到在一行代码执行处暂停了，如图 11.4 所示。

node-inspector 还在维护中，会继续开发以支持更多的 Chrome 开发者工具特性。

我们可以有两种方式来调试 Node 程序：命令行的调试器或者 node-inspector。接着我们看一下另外一个来解决性能相关的问题的工具：profiler。

技巧 92 对 Node 应用进行性能分析

性能分析主要就是为了回答这个问题：我的应用程序的运行时间都耗在哪里了？假设你有一个长时间运行的 web 服务器，当访问特定的一个路径后，发现 CPU 占用率高达 100%。乍一看，你可能会以为这个路径会有什么特殊的功能，或者也可以开启一个分析

器来让 Node 提供相关的一些信息。在这一节的内容中你会了解到如何使用分析器来获取需要的结果。

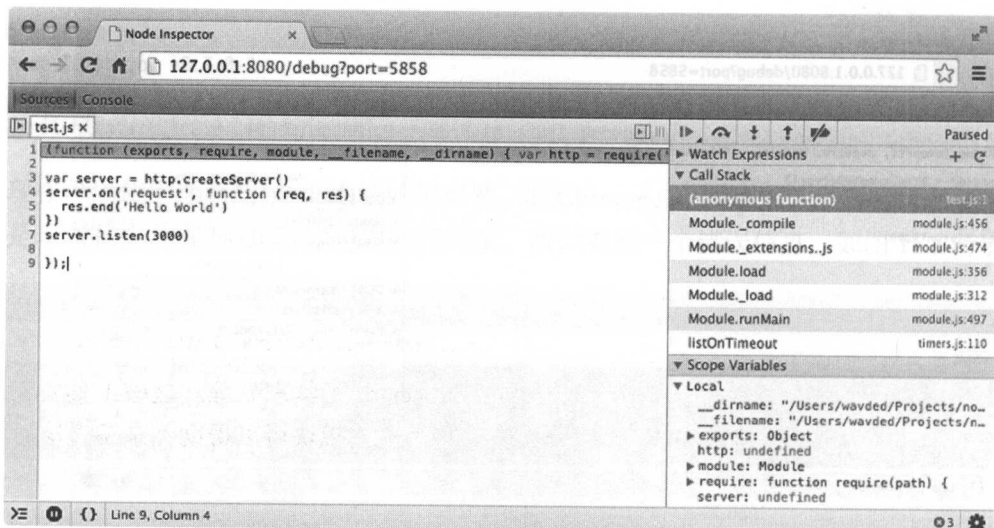


图 11.4 使用 `--debug-brk` 标识

问题

你需要找出开发的应用代码的时间开销。

解决方案

使用 `node --prof`。

讨论

Node 可以通过带上 `--prof` 参数来直接使用 V8 基础提供的统计分析器。为了更好地解释获得的数据，了解它是如何工作的非常重要。

每两毫秒，分析器会查看运行中的应用程序，并且记录函数执行的时长。这个函数可以是 JavaScript 中的函数，也可以是从 C++ 来的，或者共享库，或者 V8 的垃圾回收。分析器会把这些记录写入一个名称为 `v8.log` 的文件，这些可以由特定的 V8 `tick-processor` 程序来处理。

我们可以用一个简单的例子来看一下它是怎么工作的。这里有一个程序，来执行两个不同的任务，每两秒运行一个比较慢的计算任务，以及更加频繁地运行一个比较快的 I/O 任务：

```
function makeLoad () {
  for (var i=0;i<10000000000;i++);
}
function logSomething () {
  console.log('something');
}

setInterval(makeLoad, 2000);
setInterval(logSomething, 0);
```

我们可以这样来对程序进行分析：

```
node --prof profile-test.js
```

让它执行 10 秒左右后将其进程杀死，在同个目录下会产生一个 `v8.log` 文件。这个日志文件本身并没有多大用处。我们可以使用 `V8 tick-processor` 工具来处理。这些工具要求你在自己的机器上用源码编译安装 V8 引擎，但有些第三方模块可以帮助你跳过这些麻烦的步骤。仅仅运行下边的命令来安装即可：

```
npm install tick -g
```

这会为你的操作系统安装合适的 `tick-processor` 来查看这些数据。你可以在同一个目录下使用下边的命令来获取 `v8.log` 中更多有用的信息：

```
node-tick-processor
```

你可以获得类似下边的输出：（省略数据格式）

```
Statistical profiling result from v8.log,
(6404 ticks, 1 unaccounted, 0 excluded).
```

```
[Unknown]:
```

```
ticks total nonlib name
1 0.0%
```

```
[Shared libraries]:
```

```
ticks total nonlib name
4100 64.0% 0.0% /usr/lib/system/libsystem_kernel.dylib
211 3.3% 0.0% /Users/wavded/.nvm/v0.10.24/bin/node
...
```

```
[JavaScript]:
```

```
ticks total nonlib name
1997 31.2% 96.4% LazyCompile: *makeLoad profile-test.js:1
7 0.1% 0.3% LazyCompile: listOnTimeout timers.js:77
5 0.1% 0.2% RegExp: %[sdj%]
```

...

[C++]:

ticks total nonlib name

[GC]:

ticks total nonlib name

1 0.0%

[Bottom up (heavy) profile]:

Note: percentage shows a share of a particular caller in the total amount of its parent calls.

Callers occupying less than 2.0% are not shown.

ticks parent name

4100 64.0% /usr/lib/system/libsystem_kernel.dylib

1997 31.2% LazyCompile: *makeLoad profile-test.js:1

1997 100.0% LazyCompile: ~wrapper timers.js:251

1997 100.0% LazyCompile: listOnTimeout timers.js:77

我们看一下每一部分的意义：

- *Unknown*——这一部分的统计，分析器无法找到对应有意义的指针。这一部分会打印出来，但是没有什么实际意义，可以安全地把它忽略。
- *Shared libraries*——来自于 C/C++ 的共享类库，在这一部分也有很多的 I/O 发生。
- *JavaScript*——这通常是开发者最关心的部分，它包括了应用本身的 Node 代码以及 V8 引擎本身的原生 JavaScript 代码。
- *C++*——V8 引擎中的 C++ 代码。
- *GC*——V8 的垃圾回收。
- *Bottom up (heavy) profile*——这展示了分析器获取的耗时最高那部分数据的详细堆栈。

在例子中，我们可以发现 **makeLoad* 方法是 JavaScript 部分占比最高的，足足有 1997 个 ticks：

[JavaScript]:

ticks total nonlib name

1997 31.2% 96.4% LazyCompile: *makeLoad profile-test.js:1

7 0.1% 0.3% LazyCompile: listOnTimeout timers.js:77

5 0.1% 0.2% RegExp: %[sdj%]

这意味着它有着很重的计算任务。另外一部分值得关注的是 *RegExp: %[sdj%]*，在 *console.log* 使用的 *util.format* 中用到。

分析器的工作是告知你什么功能运行得最频繁。这并不意味着该函数执行得慢，而是会经常被调用。分析器提供的结果可以帮助你确定如何提高程序的性能。在某些情况下，找到的频繁执行的函数可能会让你大吃一惊，有时候也会正如你期望一般。分析器提供的结果可以作为一部分参考数据来帮助解决一些性能相关的问题。

性能相关的另外一个问题源于内存泄漏，很明显，影响性能结果的首先需要关注内存的使用。我们接下来看一下内存泄漏相关的处理。

技巧 93 内存泄漏的调试

在 Ajax 和 Node 出现之前，因为页面访问时长很短，JavaScript 内存泄漏相关的调试并没有投入多少努力。内存泄漏还是会发生，特别是当 Node 作为一个服务程序持续运行数天，数个星期，甚至数月。我们如何调试来找出内存泄漏的问题？在这一节中会介绍在本地或者生产环境中这一方面的内容。

问题

你需要进行程序内存泄漏相关的调试。

解决方案

使用 heapdump 或者 Chrome 开发者工具。

讨论

让我们编写一个内存泄漏的程序来说明如何使用这两个工具来调试。我们编写一个 leak.js:

```
var string = '1 string to rule them all';

var leakyArr = [];
var count = 2;
setInterval(function () {
    leakyArr.push(string.replace(/1/g, count++));
}, 0);
```

❶ 在 JavaScript 中，字符串是不可变的，我们故意每次存入一个唯一的字符串，不让垃圾回收器清理掉。

如何知道应用消耗的内存存在不断地增加？我们可以在上层进行观察，或者通过外部的一些进程监控应用来查看。或者直接输出内存使用量。为了得到准确的数据，需要在打印内存用量之前强制进行垃圾回收。我们添加以下的代码到 leak.js 文件中：

```
setInterval(function () {  
  gc();  
  console.log(process.memoryUsage());  
}, 10000)
```

为了可以使用 `gc()` 这个函数，需要在运行程序的时候使用 `--expose-gc` 参数来让 Node 暴露这个方法：

```
node --expose-gc leak.js
```

现在我们可以清晰地看到内存存在不断增长：

```
{ rss: 15060992, heapTotal: 6163968, heapUsed: 2285608 }  
{ rss: 15331328, heapTotal: 6163968, heapUsed: 2428768 }  
{ rss: 15495168, heapTotal: 8261120, heapUsed: 2548496 }  
{ rss: 15585280, heapTotal: 8261120, heapUsed: 2637936 }  
{ rss: 15757312, heapTotal: 8261120, heapUsed: 2723192 }  
{ rss: 15835136, heapTotal: 8261120, heapUsed: 2662456 }  
{ rss: 15982592, heapTotal: 8261120, heapUsed: 2670824 }  
{ rss: 16089088, heapTotal: 8261120, heapUsed: 2814040 }  
{ rss: 16220160, heapTotal: 9293056, heapUsed: 2933696 }  
{ rss: 16510976, heapTotal: 10324992, heapUsed: 3085112 }  
{ rss: 16605184, heapTotal: 10324992, heapUsed: 3179072 }  
{ rss: 16699392, heapTotal: 10324992, heapUsed: 3267192 }  
{ rss: 16777216, heapTotal: 10324992, heapUsed: 3293760 }  
{ rss: 17022976, heapTotal: 10324992, heapUsed: 3528376 }  
{ rss: 17117184, heapTotal: 10324992, heapUsed: 3635264 }  
{ rss: 17207296, heapTotal: 10324992, heapUsed: 3728544 }
```

虽然我们可以看到内存用量稳定地上升，但是从打印的内容并无法得知是怎么回事。因此我们需要获取堆使用的快照来进行对比，找出应用程序执行中是什么在变化。我们可以使用打印堆信息的第三方模块（<https://github.com/bnoordhuis/node-heapdump>）。这个 `heapdump` 模块允许我们以编程或者发送进程信号（仅支持 UNIX）的方式来获取快照。这些快照可以使用 Chrome 开发者工具来进行分析处理。

首先安装这个模块：

```
npm install heapdump --save-dev
```

然后在代码中引用这个模块，并且每隔 10 秒获取一次堆快照：

```
var heapdump = require('heapdump');  
var string = '1 string to rule them all';  
  
var leakyArr = [];  
var count = 2;
```

```
setInterval(function () {  
    leakyArr.push(string.replace(/1/g, count++));  
}, 0);  
  
setInterval(function () {  
    if (heapdump.takeSnapshot()) console.log('wrote snapshot');  
}, 10000);
```

现在，每隔 10 秒便会在当前目录下生产一个快照文件。当获取快照时，会自动调用垃圾回收。我们来让程序运行，生成两个快照文件后便终止：

```
$ node leak3.js  
wrote snapshot  
wrote snapshot
```

现在可以看一下磁盘中写入了什么：

```
$ ls  
heapdump-29701132.649984.heapsnapshot  
heapdump-29711146.938370.heapsnapshot
```

这些文件保留了各自的时间戳。数字越大便是时间越近的快照。我们可以使用 Chrome 开发者工具来加载这些文件。打开 Chrome 开发者工具，选择 Profiles 选项卡，用鼠标右键单击后选中 Load 来加载一个快照文件（详见图 11.5）。

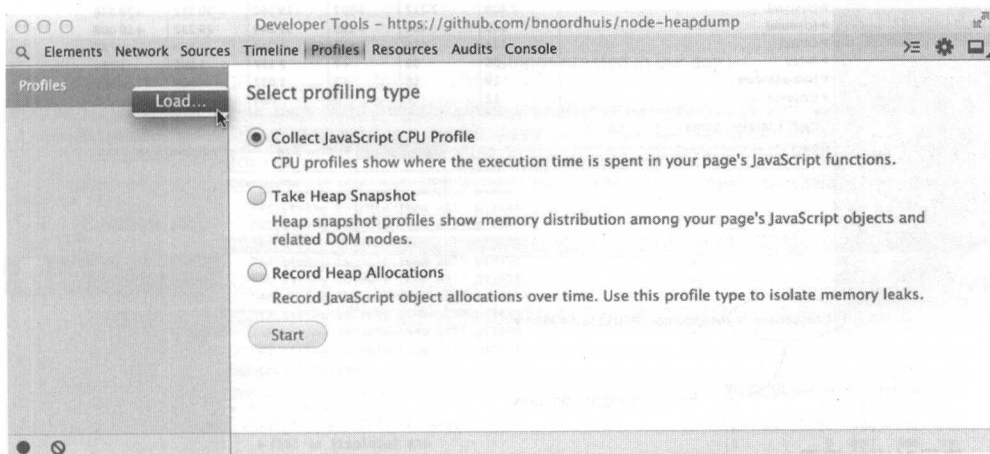


图 11.5 加载一个堆快照到 Chrome 的开发者工具

为了方便对两个快照文件进行对比，我们可以按照顺序来加载它们（见图 11.6）。

加载之后，我们可以对其进行一些研究。选中第二个后选择比较的选项。Chrome 会自动选中上一个快照来进行对比（见图 11.7）。

Node 核心和 V8 引擎中一些比较大的字符串。但当我们往下滚动时，可以看到我们的程序中生成的那些很多很多的字符串。单击其中的一个，可以看到对应的一个树状数据，包括其相关的一些对象的信息（见图 11.9）。

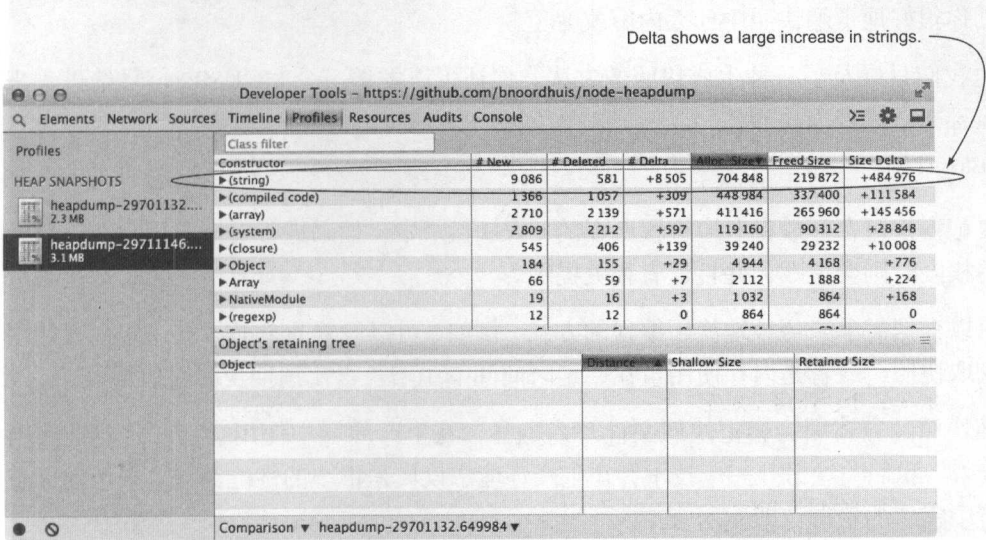


图 11.8 在两个快照间检查内存收集

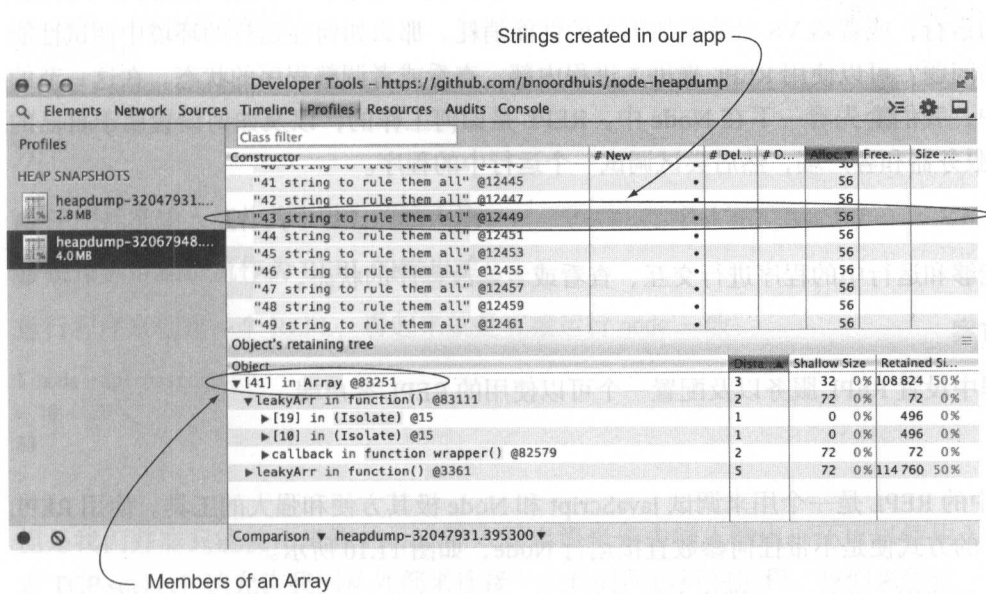


图 11.9 深入查看在内存中创建的数据类型

在这个实验中，我们明确地知道数组变量 `leakyArr` 不断地存入字符串会造成内存泄漏。但是，它却很好地为我们展示了代码和检查内存使用的工具之间的关系。作为一个开发者，你会知道源代码，在开发者工具中进入特定的代码或者模块去调试。对比的视图可以帮助你很好地了解快照中什么内容改变了。

在上边我们只提及了一种生成快照的方法。你还可以发送一个 `SIGUSR2` 信号给进程来使用 `heapdump` 来获取快照：

```
kill -USR2 1120
```

只需要记得快照会存放在对应进程所在的 `CWD` 目录中，如果进程对应的用户在该目录中无写权限时则会失败，并且没有相关提示。

你也可以根据需求来巧妙地使用编程的方式。例如，可以设置 `heapdump` 在一定内存使用后获取快照，或者是内存使用量在一定时间间隔中增长得比预期快时获取快照。

使用堆快照的方式因为会把快照文件写入磁盘，在生产环境中使用会有一定的性能损耗。接着，我们关注另外一个技术，在生产环境中仅仅是很小的性能消耗，来帮助你查看运行中程序的状态，即使用 `REPL`。

技巧 94 使用 REPL 来检测运行中的程序

给生产环境中的程序进行调试不是一个可行的方案，我们并不希望在执行调试时停止程序的运行，或者给 V8 引擎添加额外的性能消耗。那么如何在运行的环境中调试性能相关的问题？可以使用 `REPL` 来进入进程内部，查看或者调整程序的状态。在这一节的内容中，我们会先看一下在 `Node` 中，`REPL` 是如何工作的，以及如何设置属于自己的 `REPL` 服务器和客户端，然后尝试调试一个运行中的程序。

问题

希望能够和运行中的程序进行交互、查看或者修改程序的状态。

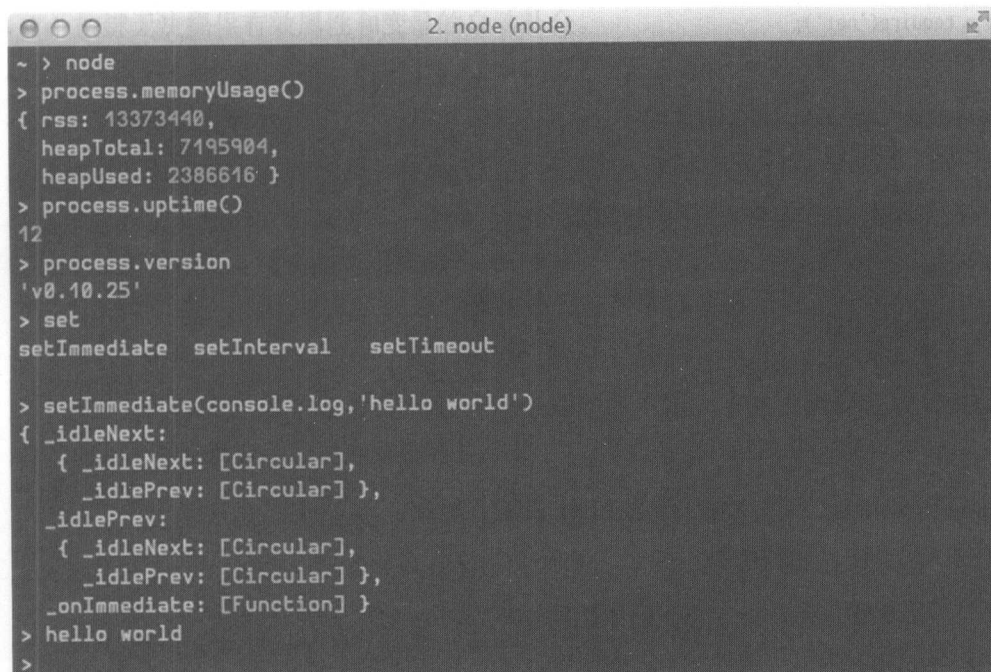
解决方案

在进程中设置 `REPL` 服务以及配置一个可以使用的 `REPL` 客户端。

讨论

`Node` 中的 `REPL` 是一个用来调试 `JavaScript` 和 `Node` 极其方便和强大的工具。使用 `REPL` 最简单的方式便是不带任何参数直接运行 `Node`，如图 11.10 所示。

你也可以使用内置的 `repl` 模块来创建自己的 `REPL`。事实上，当你输入 `node` 时，`Node` 也是使用相同的模块来运行的。我们来尝试创建自己的 `REPL`：



```
~ > node
> process.memoryUsage()
{ rss: 13373440,
  heapTotal: 7195904,
  heapUsed: 2386616 }
> process.uptime()
12
> process.version
'v0.10.25'
> set
setImmediate setImmediate setTimeout

> setImmediate(console.log, 'hello world')
{ _idleNext:
  { _idleNext: [Circular],
    _idlePrev: [Circular] },
  _idlePrev:
  { _idleNext: [Circular],
    _idlePrev: [Circular] },
  _onImmediate: [Function] }
> hello world
>
```

图 11.10 Node REPL 会话的例子

```
var repl = require('repl');
repl.start({
  input: process.stdin,
  output: process.stdout
});
```

❶ 使用标准输入流作为 REPL 的输入流。

❷ 使用标准输出流作为 REPL 的输出流。

运行程序来创建一个 REPL，看起来就是直接运行 node 一般：

```
$ node repl-basic.js
> 10 + 20
30
>
```

但是我们并非只能使用进程的标准输入输出流来作为输入输出；还可以使用 UNIX 或者 TCP socket！这允许我们从外部来连接一个长时间运行的进程。我们来创建一个 TCP REPL 的服务器：

```
var net = require('net');
var repl = require('repl');

net.createServer(function (socket) {
  var r = repl.start({
    input: socket,
    output: socket
  });
  r.on('exit', function() {
    socket.end();
  });
}).listen(1337);

console.log('node repl listening on 1337');
```

❶ 使用进来的 socket（混合流）作为 REPL 的输入输出。

❷ 当 REPL 退出时，关闭连接。

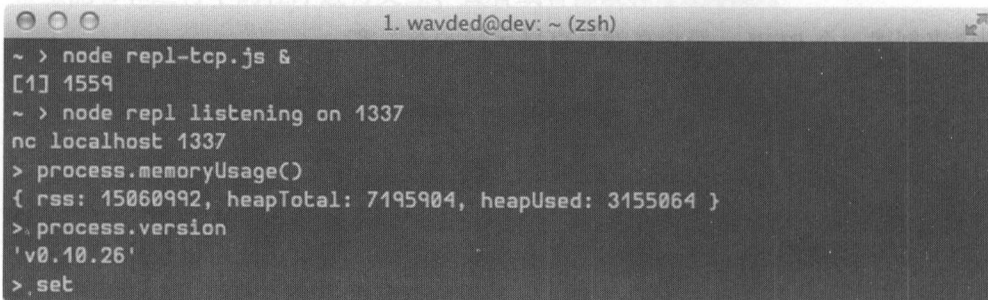
现在我们开启 REPL 服务器，它便开始监听 1337 端口：

```
$ node repl-tcp.js
node repl listening on 1337
```

接着可以使用 telnet 或者 netcat 之类的 TCP 客户端来连接。我们一个新的终端窗口来尝试一下：

```
$ nc localhost 1337
> 10 + 20
30
> exit
$
```

酷毙了是么！但是它并无法像基础的 REPL（见图 11.11）或者 node 命令一样：



```
1. wavded@dev: ~ (zsh)
~ > node repl-tcp.js &
[1] 1559
~ > node repl listening on 1337
nc localhost 1337
> process.memoryUsage()
{ rss: 15060992, heapTotal: 7195904, heapUsed: 3155064 }
> process.version
'v0.10.26'
> .set
```

图 11.11 使用 Netcat 来连接 REPL 服务器

- Tab 按键无法提供有效属性和变量的自动补全。
- 没有任何的 readline 支持，所以 Up Arrow 的按键无法查看命令历史。
- 没有颜色和加粗显示。

这种情况的原因是有两方面的。首先，repl 模块并无法决定我们是在一个 TTY（终端）会话中运行的，所以它便提供了一个最小化的接口来防止 ANSI/VT100 对颜色和格式进行转义。这些转义码对一些使用类似 Netcat 的用户来说则是无用噪音。第二，我们的客户端无法像 TTY 一样。它无法发送合适的编码来获取其他类似自动补全和历史记录这些增强。

为了调整这个情况，我们需要同时修改服务端和客户端。首先，需要发送合适的 ANSI/VT100 转移码来表示颜色和加粗等输出格式，我们需要在 REPL 的配置中添加终端的选项：

```
var net = require('net');
var repl = require('repl');

net.createServer(function (socket) {
  var r = repl.start({
    input: socket,
    output: socket,
    terminal: true
  });
  r.on('exit', function() {
    socket.end();
  });
}).listen(1337);

console.log('node repl listening on 1337');
```

❶ 配置输出和 TTY 流一致。

第二，为了获取输入的 Tab 补全和 readline，我们需要创建一个可以发送 TTY 输入源给服务器的 REPL 客户端。我们可以使用 Node 来创建：

```
var net = require('net');
var socket = net.connect(1337);

process.stdin.setRawMode(true);
process.stdin.pipe(socket);
socket.pipe(process.stdout);

socket.once('close', function () {
```

```
process.stdin.destroy();  
});
```

- ❶ 连接 REPL TCP 服务器。
- ❷ 把标准输入作为 TTY 的原始输入流。这允许 Tab 按键和 Up Arrow 按键可以如同你所期望的，在现代的终端会话中一般。
- ❸ 把标准输入的流通向 socket。
- ❹ 把 socket 的输出流通向标准输出。
- ❺ 当连接结束时，销毁输入流，来让进程可以退出。

现在我们可以拥有如同终端一般的 REPL 服务了：

```
$ node repl-tcp-terminal.js  
node repl listening on 1337
```

我们可以在另外一个终端会话中使用 REPL 客户端来连接服务：

```
$ node repl-client.js  
> 10 + 20  
30  
> .exit  
$
```

现在 REPL 会话可以像使用 node 或是基础的 REPL 时一样了。我们可以使用自动补全、浏览命令的历史记录，以及配色加强。真棒！

查看一个运行中的进程

我们已经讨论过如何使用 repl 这个模块来创建服务和使用多个客户端来连接使用。我们可以依赖这个方式舒服地在应用程序中使用 REPL 实例进行调试。现在来实践一下，使用创建的 REPL 客户端来连接服务器进行交互，来调试一个现有的程序。

首先，创建一个基础的 HTTP 服务：

```
var http = require('http');  
var server = http.createServer();  
server.on('request', function (req, res) {  
  res.end('Hello World');  
});  
server.listen(3000);  
console.log('server listening on 3000');
```

这看起来很熟悉，但是需要添加以下代码来暴露一个服务给 REPL 服务器。

```
var net = require('net');
var repl = require('repl');
net.createServer(function (socket) {
  var r = repl.start({
    input: socket,
    output: socket,
    terminal: true,
    useGlobal: true
  });
  r.on('exit', function() { socket.end() });
  r.context.server = server;
}).listen(1337);
console.log('repl listening on 1337');
```

① 允许脚本在全局的上下文和独立的上下文中执行。

② 把我们的服务实例暴露给 REPL。

有关 useGlobal 的提示

当开启的时候，无论何时你创建一个新的变量（如 `var a = 1`），它会存放到全局的上下文中（`global.a === 1`）。但是 `a` 现在也可以在待会的事件循环的函数中访问到。

我们通过在 `r.context` 上设置一个属性来暴露 `server` 给外部。我们可以暴露任何想要的东西给到 REPL 以便可以进行交互。有一个重要的点需要留意的是，我们可以覆盖已经在 `context` 的任意值。这包括所有标准的 Node 全局变量，如 `global`、`process` 或 `Buffer`。现在我们已经把 `server` 暴露出来，我们看下如何注入和调试我们的 HTTP 服务器。首先，我们开启 HTTP 和 REPL 服务：

```
$ node repl-app.js
server listening on 3000
repl listening on 1337
```

现在让我们使用 REPL 客户端来连接到服务器：

```
$ node repl-client.js
>
```

我们现在就可以获取有用的信息。例如，我们可以查看进程已经运行了多久，以及使用了多少内存：

```
> process.uptime()
```

115

①

```
> process.memoryUsage()
{ rss: 17399808,
  heapTotal: 7195904,
  heapUsed: 4146840 }
```

❶ 运行时间，单位秒。

❷ 内存用量，单位 byte。

因为我们暴露了 server 对象，所以可以通过输入 server 来进行访问：

```
> server
{ domain: null,
  _events:
    ...
  _connectionKey: '4:0.0.0.0:3000' }
```

我们看下现在有多少个连接是活动的：

```
> server.connections
0
```

显然，如果这是在生产环境中的话会更加有趣，因为只有我们自己会使用这个服务器，而且我们还没有创建连接。在浏览器中打开 <http://localhost:3000> 然后输入命令来查看值是否有变化：

```
> server.connections
6
```

❶ 浏览器/客户端的连接变化。

这已经生效了。我们来看一些更加复杂的情况。你能够想到一种使用 REPL 来记录进入到我们服务器请求数的方法么？

添加描述

REPL 强大的一个方面是可以添加描述来帮助我们理解应用运行中的一些行为。这对于处理棘手问题是特别方便的，例如这种情况：重启应用时会丢失先前的状态，我们没有办法重现问题，只能等待它再次发生。

因为我们的 HTTP 是继承于 EventEmitter 的，我们可以使用 REPL 来添加另外一个请求处理方法，它会在每一次请求时被调用，来增加我们想要的行为：

```
> var numReqs = 0
undefined
```

```
> function trackReqs (req, res) {  
  .... numReqs++  
  .... }  
undefined  
> server.on('request', trackReqs)  
  { domain: null,  
    _events:  
      ...  
    _connectionKey: '4:0.0.0.0:3000' }  
>
```

- ❶ 创建变量来存储请求数量。
- ❷ 为进来的请求创建处理方法，用于增加请求计数。
- ❸ 添加处理方法到请求事件中。

现在我们可以跟踪进来的请求了。我们在浏览器刷新几次后来看看它是否能够正确运行：

```
> numReqs  
8
```

真棒！因为我们可以访问到请求对象，所以我们可以获取请求中任意对我们有用的信息：IP 地址、头部信息、路径等。在这个例子中我们对外暴露了一个 HTTP 服务器，但是任意在你应用中有意义的对象都可以放到 `context` 中。你可能甚至想到编写一个模块在 REPL 中暴露一些常用的方法。

一些问题没办法在应用层面上解决，需要更加深层的系统反馈。一种来深入了解系统内部的方法是跟踪。

技巧 95 跟踪系统调用

了解底层系统的调用可以真正帮助你更好地了解整个平台。例如，Python 和 Node 都具备 DNS 的查找功能，但是它们在底层部分有一些不同。如果你想知道一些行为上的底层差异，跟踪工具可以帮助你。

在核心中，跟踪工具可以监控一个应用或者多个应用相关的系统底层调用（通常是 C 语言函数名称、参数和返回值），并且可以利用获取的数据做一些有趣的事情（例如日志记录或者统计）。

在生产环境中，跟踪也是相当有效的工具。如果你有一个进程占用了 100% 的 CPU 资源，并且你无法确定是什么原因导致的，那么一个跟踪工具可以帮助你查看到底层系统级别的调用状态。例如，你可能会发现这个是由于允许打开的文件溢出了，导致所有的

I/O 都被拒绝了而造成的问题。由于跟踪工具不像分析器一般占用系统性能，所以是相当有价值的工具。

问题

· 你希望知道应用运行时在系统级别程序发生了什么。

解决方案

使用专门用于操作系统的跟踪工具来获取信息。

讨论

到目前为止我们讨论过的技术点都是和操作系统无关的。而这一节的内容则是和操作系统紧密联系的。有很多不一样的跟踪工具，但是对操作系统来说则大部分是唯一的。在我们的例子中，使用 Linux 特定的工具，称为 `strace`。在 Mac OS X/Solaris 类似的工具是 `dtruss`，而在 Windows 中则是 `ProcessMonitor` (<http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>)。

一个跟踪程序本质上是转储在程序进程中系统级别的调用信息。如果你对操作系统底层不大熟悉，要准备好好学习一下了。我们将通过跟踪一个简单的应用程序，看看在操作系统级别发生了什么事情，来学习如何阅读跟踪日志。

我们先写一个最简单的程序来尝试跟踪一下：

```
console.log('hello world');
```

看起来已经足够了，我们使用下边的命令来尝试跟踪一下：

```
sudo strace -o trace.out node hello
```

程序如预期一般输出“hello world”后退出了。但是我们已经获取了系统调用的信息，存放在 `trace.out` 中。看一下这个文件：

首先在第一行中可以看到第一个有意义的调用，运行了 `/usr/bin/node`，传递的参数是 `node` 和 `hello`：

```
execve("/usr/bin/node", ["node", "hello"], [/* 24 vars */]) = 0
```

如果你曾想过为什么 `process.argv[0]` 是 `node` 和 `process.argv[1]` 是我们 Node 程序的路径，现在你会发现，底层调用就是这样的！`strace` 输出告知了我们传递的参数和返回的值。

想要知道 `execve` 更多的信息（或者其他的系统调用），我们可以直接查看主机上的 `man page`（如果可用最好），或者直接在线查看：

man execve

关于 man 命令

手册还包括了很多错误代码细节的相关帮助信息，比如，ENOENT 或者 EPERM 在操作系统上的含义。很多的错误代码可以在帮助手册上查找得到。

继续看那个文件。很多初始化的调用是在加载 libuv 需要的共享库。然后到了程序部分：

```
getcwd("/home/wavded", 4096) = 13
...
stat("/home/wavded/hello", 0x7fff082fda08) = -1
    ENOENT (No such file or directory)
stat("/home/wavded/hello.js",
    {st_mode=S_IFREG|0664, st_size=27, ...}) = 0
```

可以看到 Node 程序获取到当前的工作目录，然后执行我们的文件。注意，我们执行应用时并不带.js 扩展名，所以 Node 先尝试查找名称为“hello”的程序，找不到之后，再开始查找 hello.js。如果运行时带上.js 扩展名，那么便看不到这个状态的调用。

继续看下一个有趣的部分：

```
open("/home/wavded/hello.js", O_RDONLY) = 9
fstat(9, {st_mode=S_IFREG|0664, st_size=27, ...}) = 0
...
read(9, "console.log('hello world')\n", 27) = 27
close(9) = 0
```

这里以只读状态打开了 hello.js，并且分配了一个文件描述符。文件描述符是由操作系统分配的一个整数，为了了解后续的调用，我们需要明白，在关闭调用前，9 是分配给 hello.js 的描述符。

打开之后，可以看到使用了 fstat 来获取文件大小。然后便是按行读取文件内容。strace 的输出也给我们展示了在 buffer 中存储的内容，最后关闭文件描述符。

跟踪输出的文件内容不会显示应用程序正在运行。只能看到运行对于系统的影响是怎么样的。也就是说，看不到 V8 引擎解析或执行 console.log 代码，但是可以看到底层通过标准输出来打印结果。就像下边一样：

```
write(1, "hello world\n", 12) = 12
```

我们可以返回第 6 章看一下，每一个进程都会自动分配三个文件描述符，stdin(0)、stdout(1)、stderr(2)。我们可以看到 write 调用使用了 stdout(1) 来输出 hello world。也可以看到 console.log 给我们添加了一个新行。

trace.out 中的最后一行便是意味着程序退出：

```
exit_group(0)
```

0 代表了程序退出时的状态。这个例子程序正常运行了，所以是 0。如果我们的程序是以 process.exit(1) 或者其他状态退出了，这里可以看到返回的对应数字。

跟踪正在运行的进程

好了，我们已经使用 strace 来启动和跟踪程序，直到它退出。那么如何监控正在运行的进程呢？

这里先来获取进程的 PID：

```
ps ax | grep node
```

在这一行输出中的第一个数字便是我们要的 PID：

```
32476 ? Ssl 0:08 /usr/bin/node long-running.js
```

有了 PID 之后，我们便可以使用 strace 来监控了：

```
sudo strace -p 32476
```

这样，运行中的进程所有系统调用信息会在控制台中打印出来。

当实际应用中调试 CPU 挂起的问题时，跟踪工具将是强大的第一防线。例如，我们进程的 ulimit 溢出时，通常是 CPU 挂起，然后 open 的系统调用会一直失败。运行 strace 监控对应的进程很快便可以发现很多的 ENFILE 错误。从帮助手册中，我们可以看到错误代码的相关信息：

```
ENFILE The system limit on the total number of  
open files has been reached.
```

列出打开的文件

在这个例子中，可以使用另外一个方便的 Linux 工具——lsof，通过进程的 PID 来获取进程打开的文件列表来查看究竟打开了哪些文件。

当 CPU 负载 100% 时，我们也可以使用 strace 来找出原因，如果发现下边的信息一遍又一遍出现：

```
futex(0x7ffbe00008c8, FUTEX_WAKE_PRIVATE, 1) = 1
```

大多数情况下，这是事件循环报出的错误，你的程序很有可能哪里出现了死循环。像 node --prof 之类的工具可以帮助你定位错误。

其他操作系统的相关工具

我们看到的系统调用在不同的操作系统上可能会完全不一样。例如，在 Linux 上看到的 `epoll` 调用在 Mac OS X 中是不可能看到的，`libuv` 在 Mac 中使用 `kqueue` 来替代 `epoll`。尽管大部分操作系统都有 POSIX 方法，如 `open`，但是函数签名和错误代码可能会不一样。更好地了解开发 Node 使用的主机机器，有助于你更好地使用跟踪工具。

课外作业

编写一个简单的 HTTP 服务器并且使用跟踪工具监控。看看能否找出哪里是端口绑定，哪个是接受连接，以及哪里的响应返回给客户端。

11.3 总结

这一章中，我们关注如何调试 Node 应用。首先，将重点放在错误的处理和预防：

- 如何处理应用发生的错误？
- 如何处理应用崩溃？有设置域或者绑定 `uncaughtException` 处理事件吗？
- 有使用代码检查工具来防止异常发生吗？

然后关注如何调试特定的问题。使用 Node 中各种各样的工具和第三方模块。最重要的事情是掌握正确的工具来处理相应的问题。当出现问题时，你可以对其评估，来获得有用的信息：

- 是否需要设置断点、监测表达式，以及步进式地调试代码？使用内置的调试命令或者 `node-inspector`。
- 是否需要查看应用的时间消耗？使用 Node 内置的分析器（`node --prof`）。
- 应用是否消耗过多的内存？使用堆快照并且分析其结果。
- 是否需要在不停止程序运行的前提下，深入查看程序的运行状态，或者了解性能损耗？设置和使用 REPL 服务。
- 需要了解使用了哪些底层的系统调用吗？使用操作系统的跟踪工具。

在下一章中我们会深入了解使用 Node 来编写 web 应用。

12 生产环境中的 Node：安全地部署应用程序

本章概要

- 将 Node 程序部署到自己的服务器上
- 将 Node 程序部署到云平台上
- 管理生产环境下的包管理
- 日志
- 利用代理服务器和集群扩展 Node 应用

一旦你的 Node 程序构建好了，并且也测试好了，接着就需要发布它了。目前有一些流行的 PaaS（平台及服务）提供程序-云服务平台提供程序，譬如 Heroku、Nodejitsu 这样的平台已经让部署变得很简单，当然你可以选择将你的程序部署到自己的私有服务器上。当程序部署好了之后，接下来你要做的就是处理一些程序可能会出现的非预期的异常、程序服务中断、程序错误以及监视程序运行的性能。

本章将会讲解如何安全地发布和维护 Node 程序。我们会讲到如何将程序部署到私有主机上。主要会涉及的知识包括 Apache、nginx、WebSockets、水平扩展、自动化部署、日志以及提高性能的方法。

12.1 部署

在这节，我们将会讲解如何将你的程序部署到当前流行的云提供程序和私有服务器上。通常来说，你很有可能只会使用到其中的一种方式，这取决于你的程序的需求和你的雇主了。然而，同时掌握这两种部署的方式对你还是很有益的。比如，基于 Git 工作流的 Heroku 就已经影响了很多人部署程序的方式，大部分采用它的人都只需要掌握一点如何设置服务器的知识就可以轻松部署应用程序，而不用去向一些专门的开发运营专家寻求咨询。

接下来的第一个知识点是基于 Windows Azure、Heroku 和 Nodejitsu 的。这可能已经是目前用于部署 web 应用程序最简单的方式了，并且现在的云服务提供商都会提供一些免费的服务计划，使得部署成本变低，这样就可以很快乐地与别人分享你的杰作了。

技巧 96 将 Node 程序部署到云端

本技巧将会概述如何使用将 Node 程序在云提供服务上工作，并且会告诉你一些在生产环境下如何配置和维护应用程序的技巧。我们主要还是专注在实际的发布和维护的实操上，而不是云服务的价格和商业层面上。

你可以免费地尝试 Heroku 和 Azure 的云服务，顺带体验一下如何在云上运行你的 Node 应用程序。

问题

你已经使用 Node 构建好了一个应用程序，并且想将它部署到一台服务器，这样人们也可以使用它。

解决方案

使用 PaaS 服务提供商，比如 Heroku 和 Nodejitsu。

讨论

我们将会分别探讨 3 种云部署的平台：Nodejitsu、Heroku 和 Windows Azure。这几个平台都可以允许你部署 Node web 应用，但是在处理的时候各自还是有细微区别的。上传和配置应用程序的方式是不同，但是基本的概念其实还是一样的。

Nodejitsu 是一个很有意思的案例，因为它只为 Node 程序服务。另外，Windows Azure 支持微软自身的软件开发工具、程序语言和数据库。Azure 拥有的特性已经远超于部署和托管应用程序，比如它还有数据库和活动目录的集成功能。Heroku 则拥有一个很活跃

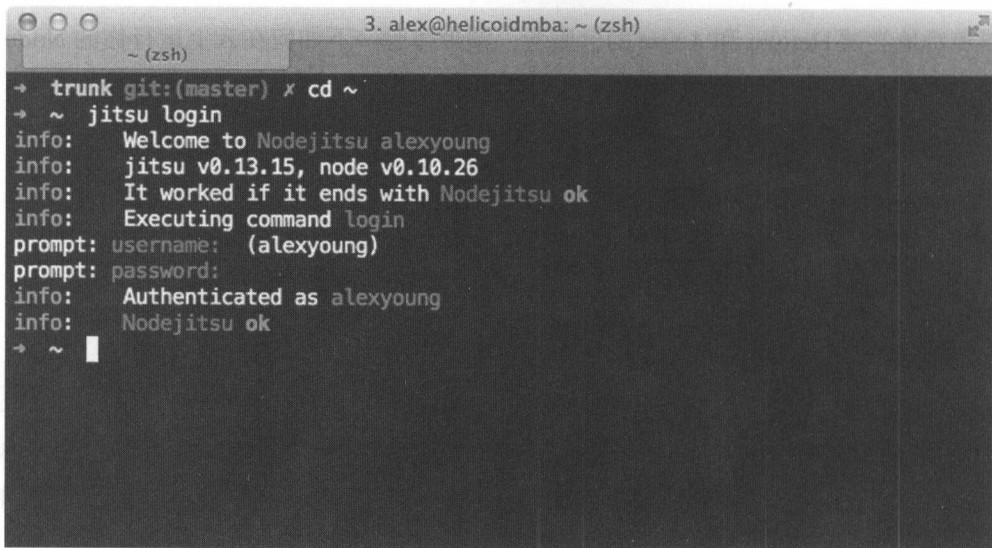
的社区, 拥有很多的合作伙伴, 你可以找到很多插件, 相比之下, Azure 更多的是提供全套服务的平台。

如果你看过这本书配套的源代码, 应该会发现 `production/inky` 下有一个简洁版的应用程序, 我们也是用这个程序来研究本章技巧的, 你可以使用这个应用程序来试验所有的云服务平台。Nodejitsu 和 Azure 的文档里包含了一些基于 Node http 模块的示例。但是你需要在 `package.json` 做一些修改来真正了解 Node 应用程序到底是怎么工作的。

首先要介绍的就是 Nodejitsu (<https://www.nodejitsu.com/>), Nodejitsu 的总部设在纽约, 在北美和西欧都有数据中心。Nodejitsu 建立于 2010 年, 由 Bloomberg Beta 基金公司提供基金。

为了进一步学习 Nodejitsu, 可以先注册一个账号, 登录到 Nodejitsu.com, 你可以不选择价格计划并且登录, 如果你只是想通过 Nodejitsu 发布一个开源项目。

Nodejitsu 已经具有一个命令行客户端, 名为 `jitsu`, 你可以通过运行 `npm install -g jitsu` 来安装它。一旦通过 `npm` 安装成功, 就可以使用 `jitsu login` 登录了, 在登录的时候需要提供用户名和密码, 在这个过程中它会将一个 API 的 token 保存到 `~/.jitsuconf` 文件中。这样你的密码就不会被保存在本地了。图 12.1 显示了这个使用终端的流程。

A terminal window titled '3. alex@helicoidmba: ~ (zsh)' showing the execution of the 'jitsu login' command. The output includes a welcome message, version information for jitsu and node, a confirmation message, and a successful authentication as 'alexyoung'. The prompt returns to '~' after the login process is complete.

```
3. alex@helicoidmba: ~ (zsh)
~ (zsh)
→ trunk git:(master) x cd ~
→ ~ jitsu login
info: Welcome to Nodejitsu alexyoung
info: jitsu v0.13.15, node v0.10.26
info: It worked if it ends with Nodejitsu ok
info: Executing command login
prompt: username: (alexyoung)
prompt: password:
info: Authenticated as alexyoung
info: Nodejitsu ok
→ ~
```

图 12.1 你可以使用 `jitsu` 命令行进行登录

输入 `jitsu deploy` 来发布一个应用。`jitsu` 命令会提示关于你的应用的相关问题, 然后设置它在一个临时的子域名中运行。如果你是使用 Express 应用, 它会自动设置 `NODE_ENV`

变量为生产环境,但是你可以在 web 界面来编辑这些设置的环境变量。事实上,web 界面可以处理大多数 jitsu 命令可以处理的事情,这意味着你没必要依赖一个开发者来做基础的维护工作,如重启应用等。

图 12.2 展示了 Nodejitsu 的 web 界面,它叫作 *WebOps*,通过它,你可以启动和停止应用程序,管理环境变量,将你的应用程序回滚到更早的版本,甚至是实时地输出日志流。

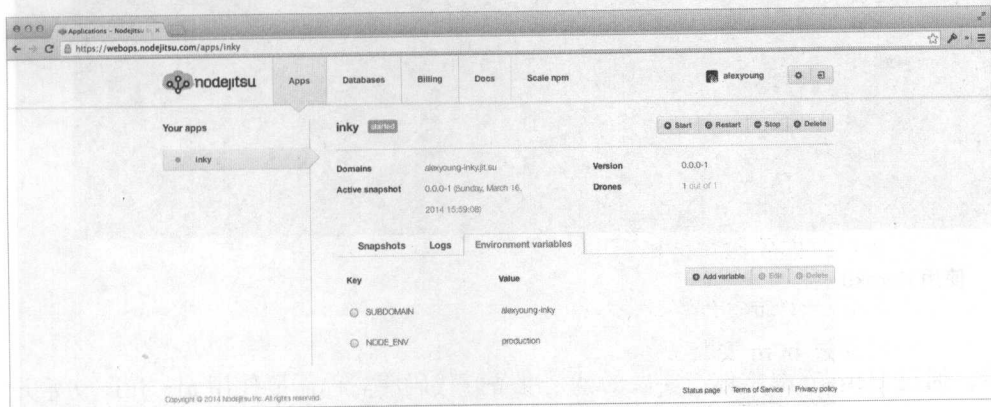


图 12.2 WebOps 的管理界面

Nodejitsu 是一个完全为 Node 应用程序打造的平台,它的部署过程很大程度上受到 npm 的影响。如果你对 npm 和 package.json 有较好的掌握,并且你需要部署的应用程序都是用 Node 开发的话,那么使用 Nodejitsu 就如鱼得水。

另外一个 Node 开发人员青睐的 PaaS 平台就是 Heroku, Heroku 支持多种编程语言和平台,包括 Node, Heroku 于 2007 年创立,现在已经被 Salesforce.com 收购了。它使用了一种基于 Ubuntu 服务器上的虚拟化解决方案。你需要登录 heroku.com 来使用 Heroku。在上面创建一个免费的账号是很容易的,甚至也可以让你生产环境下的程序使用它们提供的免费服务。只有一些必要的特性如域名别名和 SSL 都是要收费的,所以说如果你不介意使用 Heroku 的二级域名,那么你完全可以通过免费的方式来托管运行应用程序。

一旦创建了一个账号,你就需要使用 Heroku 的 Toolbelt 在本地安装 Heroku,你可以在 toolbelt.heroku.com 下载到 Heroku 的 Toolbelt。它针对 Linux、Mac OS X 和 Windows 平台上都有各自不同的安装包。一旦你安装了它,就可以通过一个叫 heroku 的客户端命令窗口来运行 heroku,你通过它创建和管理你的应用程序。在你使用它之前,必须先登录;你可以通过 heroku login 登录,大致的方式和 Nodejitsu 的 jitsu 很是相似。一旦你登录过一次之后,因为它已经存储好了一个 token,这样的话以后就不需要再登录了。图 12.3 显示了它是怎么工作的:

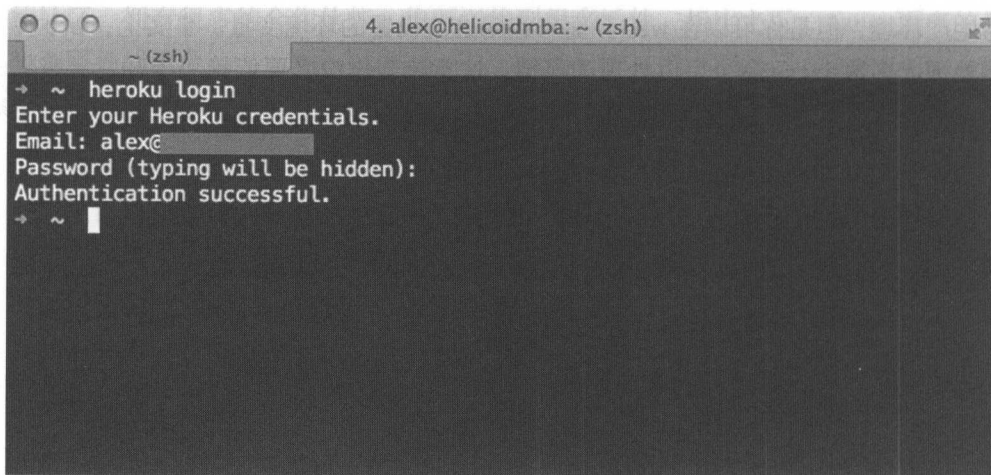


图 12.3 使用 Heroku 进行登录

下一步, 通过 Heroku 部署你需要做的就是准备好代码库。你需要使用 `git init` 初始并提交项目。如果你使用的是我们提供的代码示例, 并且你已经从 Git 里签出, 那么需要拷贝这个特定的项目, 这样就可以将它从我们的工作目录中部署出去。下面展示了整个过程的步骤:

1. `git init`
2. `git add .`
3. `git commit -m 'Create new project'`
4. `heroku create`
5. `git push heroku master`

使用 `heroku create` 命令在远端设置一个名为 `heroku` 的代码库, 上面的第一个 `git push` 将会创建一个临时的 `herokuapp.com` 的二级域名。

如果你的程序可以通过 `npm start` 的方式来运行, 那么它应该是可以工作的, 如果不可以, 那么你也可能需要为你的程序创建一个名为 `Procfile` 的文件, 它包含 `web: node yourapp.js` 的内容。这个文件列举了你的应用程序需要运行的进程——这也包括后台工作进程。

如果你使用的是一个简洁版本的程序, 并且期望设置 `NODE_ENV`, 那么需要在 Heroku 里手动设置它。具体的命令就是这样的: `heroku config:set NODE_ENV=production`, 但是注意, 这个 Nodejitsu 是自动完成的。

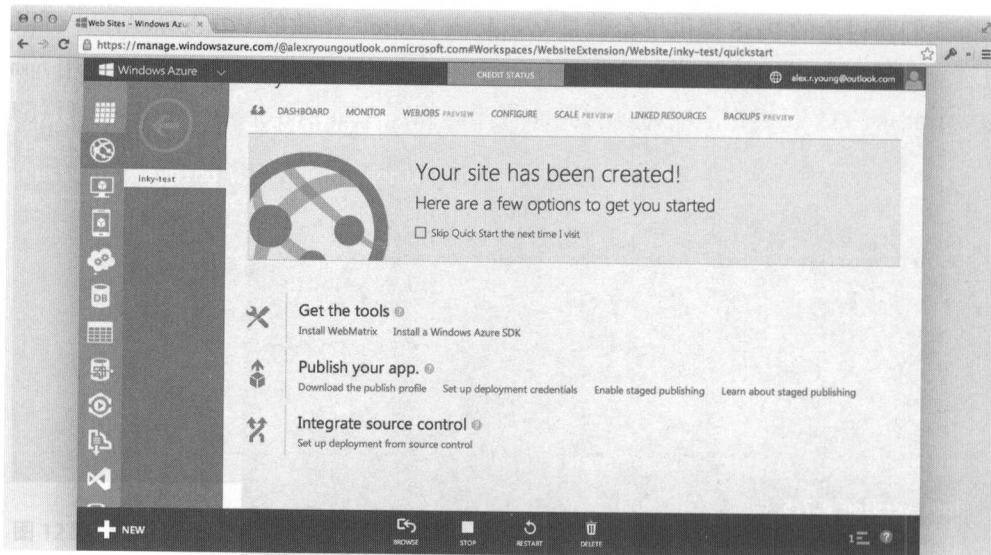


图 12.5 Azure 在创建完一个网站之后的界面

云配置

所有的 PaaS 平台貌似都有它们各自的配置应用程序的方法,当然你可以将你的配置信息保存在你的代码或者是 JSON 文件中,但是有时候将它们存储在代码库之外显得更有用处。

比如,我们正在构建一个开源的 web 应用程序,我们想将它部署到 Heroku 上,所以你就可以通过使用 `heroku config:set` 将你的数据库账户和密码信息保存在开源的代码库库之外。

单击你的应用程序名称,选择 *Set up deployment from source control*,然后在右边找到你的站点的 URL 地址。到现在为止,你应该可以选择很多云提供程序,但是我们使用 Github 来测试我们的程序。Azure 提取相关的代码,设置了一个 Node 程序——这个是和我们在前面 Heroku 里使用的简洁应用程序 (`listings/production/inky`) 是一样的,并且它一开始是可以工作的。

表 12.1 展示了如何在不同的云提供程序中获取和设置我们上面讨论过的相关配置。

尽管 Azure 的注册需求看上去也许没有 Heroku 和 Nodejitu 那么方便,但是它确实也有很多的优点:如果你正在使用的是 .NET,那么就可以使用现成的工具。同时,微软的

文档支持也十分好，它包含关于 Linux 和 Mac OS X 两个平台的安装和部署向导的相关文档（<http://www.windowsazure.com/en-us/documentation/articles/web-sites-nodejs-develop-deploy-mac/>）。

表 12.1 设置环境变量

供应商	设 置	删 除	列 表
Nodejitsu	jitsu env set name value	jitsu env delete name	jitsu env list
Heroku	heroku config:set name=value	heroku config:unset name	heroku config
Azure	azure site appsetting add name=value	azure site appsetting delete name	azure site appsetting list

不管你使用的是自己的服务器还是租用的服务器，抑或是使用廉价的虚拟主机，它们都有各自的优势。如果你想完成对主机的控制，或者是已经有用自己的服务器或者数据中心，那么你通过阅读这些文档去学习如何将 Node 部署到自己的服务器上。

技巧 97 使用 Apache 和 Ngnix 部署 Node 程序

将 Node 部署至运行 Apache 或者是 nginx 的私有服务器是完全可能的，并且我们在某些情况下也推荐你这么 做。我们将在这个技巧展示如何在 Apache 或者是 nginx 的环境下运行 Node 程序。

问题

您想在自己的服务器上运行 Node web 应用。

解决方案

使用 Apache 或者是 nginx 反向代理和一种类似于 runit 的监督服务。

讨论

当易于使用 PaaS 解决方案的时候，很多时候，我们不得不使用特定的硬件，或者是你拥有完全控制权的虚拟机。大企业通常都拥有自己的数据中心，所以，让他们再切换到另外一个服务供应商就显得很不合理。

虚拟化技术硬件改变了 web 托管方式。基于 Linux 平台的虚拟机的托管技术一度作为托管 web 应用的关键解决方案已经有很多年的历史了，类似 Amazon 可伸缩的计算云使得创建、销毁一个这样的服务变得唾手可得。

因此很有可能在某些时候, 你会面临将 Node 应用部署到你需要自己配置和维护的服务器上。如果你已经拥有基本的系统管理的经验, 那么当然可以使用你拥有的技能和软件工具。否则, 你不得不对 web 服务器的后台监控程序以及用于让 Node 程序一直保持运行的并且可以从异常中恢复的工具相当熟悉。

这个技巧我们会展示一些关于 Apache 和 nginx 的例子。它们都是 web 服务器, 但是它们各自的配置格式都存在很大的不同, 而且它们的构建方式也各不相同。图 12.6 就展示了我们在这一节里会创建的一个基础服务器架构。

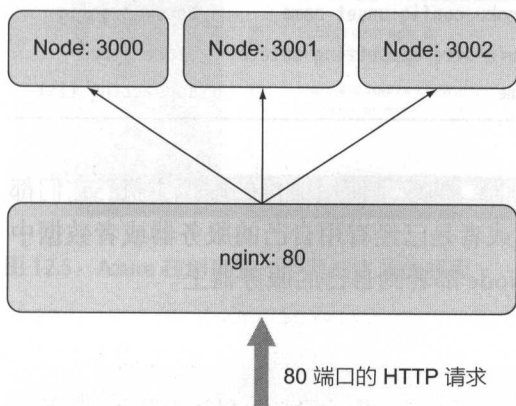


图 12.6 一个运行在 Apache 或者是 Nginx 上的 node 程序

如果只是为了让程序访问 80 端口, 没有必要运行一个 web 服务器。因为有很多方式可以让 Node 程序安全地访问 80 端口。但是我们假设你正在尝试将应用部署到一个已经部署过网站的服务器上, 同时, 有些人可能更喜欢通过使用 Apache 或者是 nginx 来部署静态资源文件。

使用 Apache 和 Nginx 的另外一个原因是可以使用它们的代理功能。下面的例子罗列了 Apache 如何做到这一点:

例子 12.1 使用 Apache 对一个 Node 程序的代理请求

```
ProxyPass / http://localhost:3000/
LoadModule proxy_module /lib/apache2/modules/mod_proxy.so
LoadModule proxy_http_module /lib/apache2/modules/mod_proxy_http.so
```

- ❶ 代理请求使用的是 3000 端口。
- ❷ 加载代理模块。
- ❸ 加载 HTTP 代理模块。

例子 12.1 里的指令应该加到 Apache 的配置文件中。为了找到正确的那个配置文件，你可以在服务器上通过输入 `apache2 -V`，并且相应地使用正确的 `HTTPD_ROOT` 和 `SERVER_CONFIG_FILE` 的值，最后将它们连接成为一条完整的命令，这样你可以找到正确的路径和文件了。可能你并不想重定向到关于 Node 应用所有的请求，所以你可以添加代理设置块：`VirtualHost`。

通过上面代码里的 3 行代码，请求到 / 将会代理到一个在端口 3000 上的监听进程^①。在这种情况下，这个进程就被假定为一个 Node 程序，这个程序正是你通过使用 `node server.js` 或者 `npm start` 运行的那个程序，但是从技术上来讲，它可以是任何的 HTTP 服务器。`LoadModule` 这个指令告诉 Apache 服务器使用代理^②，并且使用的是 HTTP 的代理模块^③。

如果你的目的在于启动或者退出一个 Node 进程，那么，Apache 就会返回一个 503 的错误代码。为了避免这样的错误发生，你需要使用一种方式使得程序一直保持运行状态。有一个办法就是通过使用 `runit` (<http://smarden.org/runit/>)。

如果你使用的是 Debian 或者是 Ubuntu 的操作系统，那么可以通过使用 `apt-get install runit` 的方式来安装 `runit`。一旦安装好了 `runit`，你可以通过创建一个 shell 脚本来启动 Node 进程。首先，为你的项目创建一个目录：`sudo mkdir /etc/service/nodeapp`，接下来，创建一个可以被脚本所使用的文件：`sudo touch /etc/service/nodeapp/run`，然后通过类似下面的方式对该文件进行编辑：

例子 12.2 使用 runit 启动一个程序

```
#!/bin/sh
export PATH=$PATH:/home/vagrant/.nvm/v0.10.26/bin
cd /home/vagrant/inky
exec npm start
```

①

②

① 设置一个 Node 的安装目录以便让 shell 可以找到。

② 切换到你目前项目的路径。

在服务器上，我们使用 `nvm` (<https://github.com/creationix/nvm>) 来管理我们安装的 node 版本。因此，将路径添加到 `$PATH` 环境变量下^①；否则，shell 无法找到 node 以及 npm 的安装目录。你必须基于自己的真实环境来设置，你可以通过 `which node` 来确定 Node 安装目录，或者也可以完全删除这一行。最后两行^②只是切换到你目前的 Node 项目目录下，然后通过使用 `npm start` 命令运行程序。

程序可以通过 `sudo sv start /etc/service/nodeapp` 被启动, 通过 `sudo sv stop /etc/service/nodeapp` 被停止。一旦 Node 进程运行起来了, 你可以通过结束进程的方式来做测试, 然后再去检查它是否可以通过运行 `runit` 来自动重启。

既然你现在已经了解了如何使用 Apache 来处理代理了, 也知道如何使得一个程序一直保持运行状态, 那接下来让我们再来介绍一下 `nginx`。`Nginx` 是一个很常用的 web 服务器, 但是在技术上它常常被应用为一个反向代理服务器, 它可以支持 HTTP、HTTPS, 以及 EMAIL。为了生成一个 `nginx` 到 `node` 程序的代理连接, 你可以使用 `Proxy` 模块, 它使用的是一个 `proxy_pass` 指令, 这一点和 Apache 的方式类似。

例子 12.3 提供了使用 `nginx` 设置的相关内容。和 Apache 一样, 你也可以把 `server` 块放到一个虚拟主机文件中。

例子 12.3 使用 `nginx` 对一个 Node 程序的代理请求

```
http {
    server {
        listen 80;

        location / {
            proxy_pass http://localhost:3000;
            proxy_http_version 1.1;
        }
    }
}
```

❶ 指向 3000 端口, 如果你使用的是多应用程序, 你可以将它改成其他端口。

如果你在同一台服务器上拥有多个应用, 那么可以使用不同的端口, 在我们的这个实例中使用 3000 端口❶。这个例子和 Apache 的示例中基本相同——告诉服务器代理指向的地址和端口号。当然, 这个例子也可以和 `runit` 结合起来。

如果你不想使用 Apache 和 `nginx`, 你可以不用 web 服务器来运行 Node 的 web 应用程序。你可以通过继续阅读后续的内容来了解通过防火墙规则或者其他技巧做到这一点。

技巧 98 在 80 端口上安全地运行 Node 程序

即使不通过类似于 Apache 这样的 web 服务器守护程序, 你也是可以通过其他办法来运行 Node 程序的。为了做到这一点, 你需要将一个外部的 80 端口转交给一个内部的、未授权的端口。在本技巧中, 我们将会展示几种在 Linux 做到这一点的方法。

问题

你不想通过使用 Apache 或者是 nginx 来运行 Node 程序。

解决方案

使用防火墙规则将一个 80 端口重定向为另外一个未授权的端口。

讨论

在大多数的操作系统中，我们需要特定权限才可以绑定到 80 端口。也就是说，如果你尝试使用 `app.listen(80)` 监听的是 80 端口而不是像我们在之前的示例中使用的 3000 端口，那么你会遇到一个这样的错误提示：`Error: listen EACCES`，原因是因为当前的用户并没有权限绑定到 80 端口。

你可以通过运行 `sudo npm start` 来绕过这个问题。但是这个却会很危险，因为理想情况下，我们还是希望 Node 程序可以运行在一个非超级用户（nonroot）之下的。

在 Linux 里，我们可以通过使用 iptables 将通信从 80 端口重定向一个更高的端口号。Linux 使用的是 iptables 来管理防火墙规则的，所以你只需要一个可以将 80 端口映射到 3000 端口的规则即可。

```
iptables -t nat -I PREROUTING -p tcp --dport \
  80 -j REDIRECT --to-port 3000
```

我们让这个变更永久生效，你需要将这个规则保存成为一个文件，这个文件会在每当网络接口启动之后就被执行。通常的办法是将这个规则保存到一个文件中，例如 `/etc/iptables.up.rules`，然后通过编辑 `/etc/network/interfaces` 来使用它。

```
auto eth0
iface eth0 inet dhcp
    pre-up iptables-restore < /etc/iptables.up.rules
    post-down iptables-restore < /etc/iptables.down.rules
```

上述代码高度依赖于操作系统，在 Debian 和 Ubuntu 的相关文档记载中，这些规则是适用的，但是可能在 Linux 的其他发布版本中，它就不一定适用。

这个技巧的一个负面影响就是，可能会将通信映射到任意一个监听那个端口进程上。解决的办法之一是可以安装 `libcap2` 来为 Node 二进制文件赋予更多的功能。

在 Debian 和 Ubuntu 上，你可以使用 `sudo apt-get install libcap2-bin` 进行安装，然后需要通过下面的命令来为 Node 二进制文件赋予更多的功能，以至于使得它可以访问有特权的端口：

```
sudo setcap cap_net_bind_service=+ep /usr/local/bin/node
```

你或许需要按照本地 Node 的实际安装地址来调整上面的命令参数, 如果你不知道安装具体目录, 可以通过运行 `which node` 来确定。使用 `libcap` 的一个负面影响是, 现在 `node` 二进制文件可以绑定到 1 到 1024 的端口, 也就导致绑定端口局限在特定的 80 端口。

一旦你为 Node 二进制文件应用了相应的功能, 这些功能将会是固定的, 除非这个文件被改变了。也就是说, 如果你升级了 Node 程序, 那么需要再次运行这个命令。

既然现在你的程序可以在一个服务器上运行了, 那么你就会想保证它永远运行下去, 为了做到这一点, 也有很多的方法, 下面的技巧就会展示使用 `runit` 和 `forever` 技术来做到这一点。

技巧 99 保持 Node 进程一直运行

任何程序都不可避免会出现崩溃的可能, 每当这个发生都是很不幸的事情。重要的是我们如何处理这种故障失败——是在失败的时候可以得到通知, 并且程序应该以一种得当的方式自动恢复。在本技巧中, 我们就会讨论如何保持 Node 程序在任何情况下都持续地运行。

问题

你的程序在半夜崩溃了, 客户必须要等到你重新启动程序之后才可以继续使用程序的服务。

解决方案

使用一个监控进程自动重启 Node 程序。

讨论

有两种主要的方法来保持一个 Node 程序持续运行: 服务监督管理和一个用于管理其他 Node 程序的 Node 程序。第一种方法是一种针对特定操作系统的通用技术。你已经在技巧 97 接触了 `runit`, `runit` 支持服务监督管理, 也就是说当一个进程停止运行的时候, 它可以检测到, 并且将进程重启。

另外一种监控管理就是使用 `Upstart` (<http://upstart.ubuntu.com/>)。如果你是使用 Ubuntu 的用户, 你可能听过 `Upstart`, 为了使用它, 需要使用一个配置文件来描述你需要管理的 Node 程序。例子 12.4 展示了一个你可以参照并为你的服务器修改的示例——它将会被存储在 `/etc/init/nodeapp.conf`, 这里 `nodeapp` 指代你实际应用程序的名字。

例子 12.4 使用 Upstart 管理一个 Node 程序

```
#!/upstart
```



```
description "ExampleApp"
author      "alex"

env PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin ❶

respawn

start on runlevel [23] ❷

script
  export NODE_ENV=production
  exec /usr/bin/node /apps/example/app.js 2>&1 >> /var/log/node.log ❸
end script
```

- ❶ 你可以按需根据应用更改 PATH。
- ❷ 这会使得程序会在级别 2 和级别 3 上启动。
- ❸ 这个命令用于运行程序。

如果程序因为某种原因而崩溃，那么这个配置文件则用于让 Upstart 重新将程序“复活”（<http://upstart.ubuntu.com/wiki/Stanzas#respawn>）。在该配置文件中，设置了 PATH 环境变量❶，如果你在终端里敲入 `echo $PATH`，那么得到的结果将与该文件里设置的很相似。然后，它还规定了程序必须运行在级别 2 和级别 3 上❷——通常当网络监控被启动之后，程序都在级别 2 上运行。

运行级别

UNIX 系统处理运行级别会根据不同的厂商而存在差异。Linux 标准基本规范表述了运行级别 2 作为多用户模式，运行级别 3 作为网络状态下的多用户模型；在 Debian 系统中，2~5 级别都用于控制台登录和显示器管理的多用户模式。然而，在 Ubuntu 中，级别 2 则是用于网络状态下的图形用户模式。所以在使用 Upstart 之前，你应该检查下操作系统是如何实现运行级别的。

Upstart 脚本 stanza 允许你包含一些简短的脚本，这就意味着你可以像在生产环境下设置 `NODE_ENV` 一样，程序本身通过使用 `exec` 指令被执行。我们也引入一些日志支持，通过将标准输出和标准错误重定向到一个日志文件当中❸。

Upstart 的启动设置比 `runit` 要做更多的工作，但是我们在生产环境下使用的这 3 年来从来没有出现过问题。这两个工具相比传统的 `stop/start` 的初始化脚本不管是在维护和设置上都要更加简单，然而还有另外一种技术可以考虑：用于监控其他 Node 程序的 Node 程序。

Node 进程管理程序通过使用一个小程序来保证另外一个 Node 程序能够持续地运行下去。这个程序很简单，因此相对复杂的程序而言，它遇到崩溃的可能性就很小。其中一个很知名的模块就是：forever (<https://www.npmjs.org/package/forever>)，它可以被用作一个命令行程序，也可以在程序中被其他程序程式地调用。

大部分人通过使用命令行来使用它。基本的用法就是 `forever start app.js`，这里的 `app.js` 就是我们的 web 应用程序。除此之外，该命令其实还有更多的可选参数，不止于此，它可以用于管理日志文件，甚至用于包装你的程序，这样一来，它就像是一个守护程序。

为了让程序作为一个守护程序来启动，可以使用诸如下面一样的选项：

```
forever start -l forever.log -o out.log -e err.log app.js
```

上面的命令会启动 `app.js`，同时创建一些额外的文件：一个用于存储当前活动进程 PID 的文件、一个日志文件和一个错误日志文件。一旦程序运行起来，你也可以通过运行下面的命令优雅地停止正在运行的程序：

```
forever stop app.js
```

Forever 可以被任何一个 Node 程序所使用，但是它通常被用作一个让 web 应用程序保持长时间运行的一个工具。它命令式的接口使得它可以很容易和其他的 UNIX 程序一起使用。

使用 WebSockets 来部署程序会带来一系列的独特的需求。它会使得你在使用 PaaS 提供程序的时候变得困难，因为它们会终止那些能够持续一段时间的请求。如果你正在使用 WebSockets，那么可以在阅读接下来的技巧中使得你的设置可以在生产环境中更好地工作。

技巧 100 在生产环境中使用 WebSockets

Node 中非常适合使用 WebSockets 技术——同一个进程可以同时为标准 HTTP 请求和更新的 WebSockets 协议请求服务。但是，你到底应该如何在生产环境中部署使用 WebSockets 技术的程序呢？继续阅读下面的知识点来发现如何在 web 服务器和云提供程序中来做到这一点。

问题

你想在生产环境中使用 WebSockets。

解决方案

确保你正在使用的服务提供程序和你的代理服务器支持 HTTP 的 Upgrade 头信息。

讨论

WebSockets 是一项很有趣的技术，但是它还是没有引起托管服务提供商足够的重视。Nodejitsu 是第一个支持 WebSockets 的 PaaS 解决方案提供商，它使用的是 node-http-proxy (<https://github.com/nodejitsu/node-http-proxy>) 来做到这一点的。几乎所有的解决方案都会引入代理。为了理解这一点，我们需要搞清楚 WebSockets 到底是如何工作的。

HTTP 协议其实是一种无状态的协议，也就是说，客户端和服务端所有的交互都被建模成为请求和相应的模式，而这些请求和相应都保存了各自所需的状态。这种级别的封装就导致了现代客户端/服务端 web 应用程序的设计。

这种协议的缺点就是，它不支持长时间的全双工通信。现在有很多的基于 TCP 协议的全双工通信的应用，如下面一些突出的典型应用：视频流和会议，实时消息，游戏。随着 web 浏览器慢慢演变成支持更加丰富、更加高端的程序，我们自然也就想通过 HTTP 协议来模拟实现这样类型的程序。

WebSocket 协议用于支持类似于 TCP 的长接。它通过客户端和服务端使用标准的 HTTP 握手来确定是否支持 WebSockets。为了支持这样的机制，有一个新的 HTTP 头信息叫 Upgrade。因为 HTTP 的客户端和服务端经常沉浸在很多的非标准的头信息当中，服务端即使不支持 Upgrade 也可以——只是客户端将不得不回退到过时的 HTTP 轮询模式。

因为服务器端不得不艰难地处理 WebSocket 连接，明智的做法就是有效地运行两台服务器。在一个 Node 程序中，我们经常会为标准的 HTTP 请求用一个 http.listen，以及另外一个“内部”的 WebSocket 服务器。

在技巧 97，你已经见识了如何在 Node 中使用 nginx。在示例中，通过使用代理将请求从 nginx 传递到 Node 进程，这样就使得 Node 进程可以绑定到不同的端口而不只是 80 端口。通过使用相同的技巧，你可以使 nginx 支持 WebSockets。下面展示了一个典型的 nginx 配置文件 nginx.conf。

例子 12.5 在 nginx 中添加对 WebSockets 的支持

```
http {
    server {
        listen 80;

        server_name example.com;

        location / {
            proxy_pass http://localhost:3000;
            proxy_http_version 1.1;
```

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection 'upgrade';
proxy_set_header Host $host;
proxy_cache_bypass $http_upgrade;
}
}
}
```

❶ 支持 Upgrade 的头信息。

添加 `proxy_http_version 1.1`、`proxy_set_header Upgrade` ❶ 使得 nginx 过滤 WebSockets 请求到 Node 进程。上面的这个实例也会忽略 WebSockets 请求的缓存。

我们在前面已经提到过，Nodejitsu 支持 WebSockets，那么 Heroku 支不支持呢？其实也是支持的，你现在需要作为一个插件来启动，也就是说，你可以运行这样的 heroku 命令：

```
heroku labs:enable websockets
```

Heroku 的 web 服务器通常情况下会终止掉那些超过 75 秒的请求，但是如果你启用了上面这样一个插件，这就意味着请求头会带有一个 Upgrade 的头信息，这样只要在网络运行的情况下，请求就会保持运行状态。

很多时候你也许不能够轻松地使用 WebSockets。一个简单的例子就是当你使用的是老版本的 Apache，这个版本是不支持代理模块的。在这种情况下，你最好在一切事情开始之前使用并运行一个代理服务器。

HAProxy (<http://haproxy.1wt.eu/>) 就是一个灵活的代理服务器。它的使用方法和 nginx 十分类似，它也是基于事件驱动的，所以它在 Node 社区里被广为接受。如果你正使用的是老版本的 Apache，那么你可以基于不同的选项，如 URL 或者 header 之上，将 web 请求代理到 Apache 或者是 Node。

如果你想在 Debian 或者是 Ubuntu 上安装 HAProxy，那么你可以使用 `sudo apt-get install haproxy`。一旦这个被设置好了，那么你将需要编辑 `/etc/default/haproxy`，并且设置 `ENABLED=1`，这是因为在默认的设置中附带了这个设置属性，所以它也可以在默认情况下被禁用。例子 12.6 是一个配置示例，它可以路由请求到一个运行在 3000 端口上的一个 Node 的 web 应用程序，但是它可以在外部通过 80 端口被访问到。

例子 12.6 在 Node 应用程序中使用 HAProxy

```
frontend http-in
  mode http
  bind *:80
```

```
timeout client 999s
default_backend node_backend

backend node_backend
  mode http
  timeout server 86400000
  timeout connect 5000
  server io_test localhost:3000
```

1

2

- 1 允许 WebSocket 连接长时间保持活动状态。
- 2 所有的 HTTP 请求都将会被路由到你的 Node 程序上。

它可以使得 WebSockets 能够正常工作,我们已经使用了一个长时间的超时,所以 HAProxy 不会关闭 WebSockets 的连接,也就是我们通常所说的持久 (long-lived) 1。如果你运行一个监听 3000 端口的程序,然后通过运行 `sudo /etc/init.d/haproxy restart` 来重启 HAProxy,这样一来你的程序应该可以在 80 端口上能够被访问到。

你可以通过使用表 12.2 中的信息来找到适合你应用程序的 web 服务器。

表 12.2 服务器选项比较

服务器	特 性	最佳适用于
Apache	<ul style="list-style-type: none">快速资源服务有效地和已建立的 web 平台 (PHP, Ruby) 工作拥有很多的类似于代理、URL 重写的模块虚拟主机	程序可能也就在服务器上了
nginx	<ul style="list-style-type: none">基于事件驱动的架构, 非常快速配置简单代理模块可以很好地支持 Web Sockets虚拟主机	当你想托管静 Node 应用的同时想托管静态站点, 但是你还没有设置 Apache 或者一台遗留的服务器
HAProxy	<ul style="list-style-type: none">基于事件驱动的架构, 很快速能够路由到同一台机上的其他的 web 服务器上很好地支持 WebSockets	扩大到高流量站点的集群或者是复杂异构设置
Native Node proxy	<ul style="list-style-type: none">复用你的 Node 编程知识灵活	当你拥有一个具有优秀 Node 编程技巧的团队的同时, 并想扩大规模

哪一个服务器才是最适合我的呢?

在这一章, 我们并没有过多地讲述如何对服务器做一个最佳选择——我们主要关注在 UNIX 服务器上的 Apache 和 Nginx。尽管如此, 在它们之间做出选择是一件很困难的事情, 我们提供了表 12.2 的内容供你参考。

通过命名它们与后端的 backend 指令, 你的 HAProxy 设置可意识到多个“后端”。在例子 12.7 中, 我们只有一个 node_backend。也可以运行 Apache, 并将某些请求路由到基于域的名称。

```
frontend http-in
  mode http
  bind *:80
  acl static_assets hdr_end(host) -i static.manning.com

backend static_assets
  mode http
  server www_static localhost:8080
```

如果你已经拥有一组 Apache 的虚拟主机, 上面这段代码特别适用——也许像服务静态资源、博客、网站一样——你想在同一台服务器上添加 Node。Apache 可以设置监听不同的端口, 所以 HAProxy 可以放置于服务器的前端, 然后将请求路由到 3000 端口, 并且现有的 Apache 站点运行在 8080 端口。通过使用 Listen 8080 指令, Apache 允许你改变端口。

你可以使用同样的 acl 选项在基于 URL 之上来路由 WebSockets, 比如说, 你想将 WebSocket 服务器挂载在 Node 应用程序的 /chat 下, 可以使用一个特定的服务器实例来处理 WebSockets, 并且根据条件通过 path_beg 来使用 HAProxy。下面的例子展示如何实现这一点:

例子 12.7 使用 HAProxy 和 WebSockets

```
frontend http-in
  mode http
  bind *:80
  acl is_websocket hdr(Upgrade) -i WebSocket

  acl is_websocket path_beg -i /chat

  use_backend ws if is_websocket
  default_backend node_backend
```

①

②

```
backend node_backend
  mode http
  server www_static localhost:3000
```

```
backend ws
  timeout server 600s
  server ws1 localhost:3001
```

- ❶ 检查 WebSocket 头信息。
- ❷ 检查 WebSocket 的路径是否已经被占用。

HAProxy 可以基于很多参数匹配请求。在这里我们使用的是 `hdr(Upgrade) -i WebSocket` 来测试一个 Upgrade 头信息是否已经被使用过❶。正如你已经看到的，这描述的就是一个 WebSocket 握手。

通过使用 `path_beg`，并且使用 `acl is_websocket` 来匹配路由❷，现在你可以基于前缀表达式 `if is_websocket` 来路由请求。

所有的这些 HAProxy 选项可以被结合起来用于将请求路由到你的 Node 应用程序、Apache 服务器以及制定的 WebSocket Node 服务器。这就意味着，你可以从一个完全不同的进程甚至是另外一个内部的 Web 服务器运行 WebSockets 应用程序。HAProxy 对于扩展 Node 程序是一个非常好的选择——你在多个服务器上可以运行多个应用程序实例。

HAProxy 提供了一个 `weight` 选项用于允许你通过往 backend 添加 `balance roundrobin` 实现 *round-robin* 负载均衡。

起初可以不用 `nginx` 或在它之前的 HAProxy 来部署程序，但是当一切就绪之后，你可以使用代理来扩展规模。如果现在没有性能问题，那么值得注意的是，代理可以做到类似将 WebSocket 路由到不同的服务器以及处理 *round-robin* 负载均衡一样的事情。如果你已经有一台使用 Apache 2.2.x 的服务器，但是它和代理的 WebSockets 不兼容，然后你可以在 Apache 之前放弃 HAProxy。

如果你正在使用 HAProxy，那么你将不得不使用一种类似于 `runit` 或者是 `Upstart` 一样的监控守护程序来管理 Node 进程，它已经被证实为一种非常灵活的解决方案。

另外一种我们还没有讨论介绍的方法就是把一个你的 Node 程序放在一个轻量级的 Node 程序之后，使得它成为一个它自己的代理。实际上，PaaS 提供程序 `Nodejitsu` 背后就是这样运行的。

选择一个正确的服务器架构体系仅仅是成功的 Node 应用程序的第一步。你应该还要考虑性能和扩展性。下面的 3 个技巧我们将会介绍并给出一些关于缓存、Node 程序的运行集群的建议。

12.2 Node 程序的缓存和扩展性

在这一节, 我们主要介绍关于一次性同时运行多个 Node 程序副本的知识, 与此同时, 我们也会介绍一些关于缓存的技术细节。试想, 如果我们可以让客户端做更多的事情, 何乐而不为呢?

技巧 101 HTTP 缓存

即使 Node 程序被认为已经是高性能的 web 应用程序, 然而我们仍然可以拥有提高程序性能的一些方法。缓存技术就是其中的方法之一, 你应该在部署程序之前就将缓存技术考虑进来。在本技巧中, 我们就会讨论 HTTP 缓存背后的一些技术。

问题

你想减少向程序发送请求的时间。

解决方案

检查并保证正确地使用 HTTP 缓存。

讨论

现代 web 程序可能会很大, 像图像资源、字体、CSS、JavaScript 和 HTML 都会归并为可怕的负载, 这些负载伴随着多个 HTTP 请求。即使使用最好的压缩工具, 下载量仍然可能达到兆级。为了避免每次用户在站点上执行操作都必须进行等待, 最好的解决办法就是去除下载任何东西的需求。

浏览器将内容缓存到本地, 并且可以通过检查缓存来决定是否需要下载资源。这个过程主要由 *HTTP cache header* 和条件性请求来控制。在该技巧, 我们将会介绍 *cache headers* 并解释它们是怎么工作的, 所以当你在通过使用类如 *Inspector* 这样调试工具监视程序服务响应的时候, 将会期望知道使用的是什么样的 *Caching headers*。

主要的两个缓存头信息是: *Cache-Control* 和 *Expires*。*Cache-Control* 用于让服务器指定一个指令来控制资源是以何种方式缓存的。基本的指令如下所示:

- *public* ——缓存于浏览器以及任何服务器和浏览器之间的中间代理。
- *private* ——只允许浏览器来缓存资源。

- no-store —— 不对资源进行缓存（但是一些客户端可以通过特定的条件对资源进行缓存）。

如果想要了解所有的 Cache-Control 指令列表，可以参考超文本传输协议 1.1 规范（<http://www.w3.org/Protocols/rfc2616/rfc2616.html>）。

Expires 头信息用于告诉浏览器什么时候替换本地的资源。这个日期应该是 RFC 1123 格式：Fri, 03 Apr 2014 19:06 BST。超文本传输协议 1.1 规范规定不要使用超过一年的日期，不要使用太长的日期设置，因为这个行为是未定义的。

这两个头信息运行服务器告诉客户端什么时候应该对资源进行缓存。大部分的 Node 框架会为你设置这些头信息。譬如，Connect 的一个静态服务中间件，就会设置 maxAge 为 0，用于表示需要执行缓存的重新验证。如果你在调试时，在浏览器里监视网络控制面板，你应该可以看到 Express 服务静态资源使用的是：Cache-Control: public, max-age=0，以及一个基于文件日期的 Last-Modified 日期。

Connect 的 static 中间件在 send 模块当中通过使用 stat.mtime.toUTCString 来取得上次文件的修改时间。浏览器会发送一个标准的 HTTP 的 GET 请求，这个请求使用的是两个额外的请求头 If-Modified-Since 和 If-None-Match 来获取资源。Connect 将会根据文件的修改日期来检查 If-Modified-Since，并且以一个 HTTP 304 的代码作为响应，这个过程都取决于修改日期。像这样的一个 304 的响应是没有主体内容的，所以浏览器能够条件性地使用本地的内容，而不是再次下载资源。

图 12.7 展示了一个高层次的基于浏览器角度的 HTTP 缓存概述。

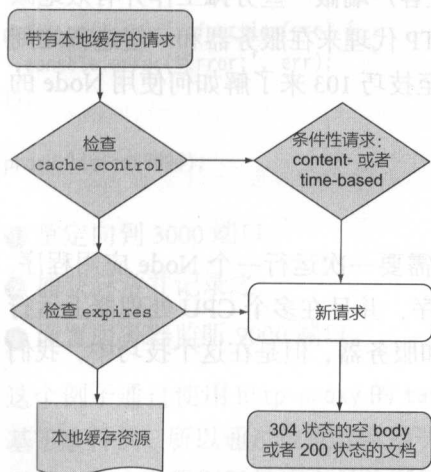


图 12.7 浏览器要么使用本地的缓存要么基于前面请求的头信息发送一个条件性的请求

条件式的缓存对于大型可能发生改变的资源是非常好的, 因为为了发现一个资源是否需要下载而发送一个 GET 请求的代价不高。这也就是所谓的“基于时间的条件请求”, 也有“基于内容的条件请求”, 即用于判断一个资源是否变化了来作为发送请求与否的依据。

“基于内容的条件请求”需要使用到 ETags。ETag 是 *entity tag* 的简称。它允许服务器基于它们的内容缓存来验证资源。Connect 的静态中间件生成的 ETags 如下:

```
exports.etag = function(stat) {  
  return '"' + stat.size + '-' + Number(stat.mtime) + '"';  
};
```

相比之下, Express 为动态内容生成 Etags——这个通常使用 `res.send` 发送内容, 就像一个 JavaScript 对象或者一个字符串:

```
exports.etag = function(body){  
  return '"' + crc32.signed(body) + '"';  
};
```

第一个例子使用的是文件的修改时间和大小来创建一个哈希值。第二个方法使用一个基于内容的哈希函数。这两种技术都像浏览器发送基于内容的标签 (tags)。但是它们都在资源类型的基础之上做了性能优化。

开发人员为了使得静态服务器变得尽可能快, 还是面临着压力的。如果你使用的是 Node 内置的 `http` 模块, 你不得不考虑素有的这些缓存头信息, 然后通过 ETags 的生成来做优化。这样就是为什么建议使用类似 Express 这样的模块——它将会处理基于敏感行为所需头信息的所有细节, 因此你就可以只需要关注程序的开发就可以了。

缓存是一种提高性能的优雅方式, 因为它通过使用让客户端做一些分摊工作并有效地减少了流量。另外一个选项就是使用基于 Node 的 HTTP 代理来在服务器和一组进程之间进行路由。继续读下面的内容, 或者你可以直接跳至技巧 103 来了解如何使用 Node 的群集模块管理多个 Node 进程。

技巧 102 为程序的路由和扩展使用 Node 代理

本地开发是一个相对简单的过程, 因为你通常都只需要一次运行一个 Node 应用程序。但是生产环境下的服务器可以同时托管多个应用程序, 并且在多个 CPU 处理器上运行同一个应用以提高性能。到现在, 我们讨论了 web 和服务器, 但是在这个技巧中, 我们将会关注纯 Node 服务器。

问题

你想使用一个纯 Node 解决方案来托管多个应用程序, 或者扩展一个应用程序。

解决方案

使用代理服务器模块，例如，Nodejitsu 的 http-proxy 模块。

讨论

本技巧将会展示如何使用 Node 程序来路由通信，这个和技巧 100 当中的代理服务器的例子很是相似，因此你可以将类似的概念应用到 HAProxy 或是 nginx。但是，通过在代码中阐述路由逻辑可能会比在设置文件当中显得更加容易。

同样，你在本书的前面已经学习到，Node 程序式以单进程的方式运行，也就是说，它通常不能够利用一个拥有多个 CPU 或者是多核处理器的现代机器。因此你可以使用这里的技术来基于你的生产需求进行路由通信，同时，做到运行多个程序的实例，这样程序就可以充分地利用服务器资源，减少相应延迟，最后客户得到满意。

Nodejitsu 的 http-proxy (<https://www.npmjs.org/package/http-proxy>) 是一个轻量级的、对 Node 内置的 http 核心模块的一个包装模块。如果你已经看过前面章节我们关于 Node web 开发的例子，你应该会熟悉它的基础用法。下面的例子就是一个简单的代理实例，它用于将通信重定向到另外一个端口。

例子 12.8 使用 http-proxy 将通信重定向到另外一个端口

```
var httpProxy = require('http-proxy');
var proxy = httpProxy.createProxyServer({
  target: 'http://localhost:3000'
});

proxy.on('error', function(err) {
  console.error('Error:', err);
});

proxy.listen(9000);
```

❶ 重定向到 3000 端口。

❷ 捕获异常并记录之。

❸ 设置服务器监听 9000 端口。

这个例子通过使用 http-proxy 的 target 选项将通信重定向到 3000 端口❶，这个模块是基于事件的，所以通过设置异常监听可以处理异常❷。代理服务器本身监听的是 9000 端口❸，在这里我们暂且用它是为了简单地运行程序——80 端口在生产环境中会被使用。

传给 `createProxyServer` 的选项信息可以定义其他的路由逻辑，如果设置的是 `ws: true`，那么 `WebSockets` 将会被单独地路由，也就是说你可以创建一个代理服务器，用它将 `WebSockets` 路由到一个应用程序和其他任何地方的一个标准请求。下面的例子展示如何将 `WebSockets` 请求路由到一个单独的应用程序：

例子 12.9 独立地路由 `WebSockets` 连接

```
var http = require('http');
var httpProxy = require('http-proxy');

var proxy = new httpProxy.createProxyServer({
  target: 'http://localhost:3000'
});

var wsProxy = new httpProxy.createProxyServer({
  target: 'http://localhost:3001'
});

var proxyServer = http.createServer(function(req, res) {
  proxy.web(req, res);
});

proxyServer.on('upgrade', function(req, socket, head) {
  wsProxy.ws(req, socket, head);
});

proxyServer.listen(9000);
```

❶ 为 `WebSockets` 创建另外一个代理服务器。

❷ 监听升级事件；然后使用 `WebSocket` 代理而不是标准的 `web` 请求代理。

这个例子创建了两个代理服务器：一个用于 `web` 请求，另一个用于 `WebSockets` ❶，当一个 `WebSocket` 被初始化的时候，主要的面向 `web` 的服务器发出一个 `upgrade` 的事件，它被得以解析使得请求可以被路由到任何地方 ❷。

这个技巧可以被扩展到根据你喜欢的任意规则来路由通信——如果你可以从一个请求对象中推理到一些东西，那么你就可以相应地路由通信。我们可以使用同样的思路来将通信映射到多台机器上。这使得你可以创建一群服务器，它们用于帮助扩展应用程序。下面的例子将会展示代理到多个服务器的代码：

例子 12.10 使用一台服务器的多个实例来扩展应用

```
var http = require('http');
```

```
var httpProxy = require('http-proxy');

var targets = [
  { target: 'http://localhost:3000' },
  { target: 'http://localhost:3001' },
  { target: 'http://localhost:3002' }
];

var proxies = targets.map(function(options, i) {
  var proxy = new httpProxy.createProxyServer(options);
  proxy.on('error', function(err) {
    console.error('Proxy error:', err);
    console.error('Server:', i);
  });
  return proxy;
});

var i = 0;
http.createServer(function(req, res) {
  proxies[i].web(req, res);
  i = (i + 1) % proxies.length;
}).listen(9000);
```

❶

❷

❶ 为每一个应用实例创建一个代理。

❷ 使用 round-robin 代理请求。

这个例子使用了一个包含一组关于每一个代理服务器选项信息的一个数组，然后为每一个应用都创建了一个代理❶，然后所以你需要做的就是创建一个标准的 HTTP 服务器并将这些请求映射到每一个服务器❷。这个例子使用的是一个基本的 round-robin 实现——在每一个请求之后，计数器都会自增，所以下一个请求就会被映射到不同的服务器。你可以简单对这个例子进行修改，然后重新配置，使之映射到不同数量的服务器上。

在一台拥有多核处理器或者是多个 CPU 的机器上，使用类似这样的请求映射技术是十分有用的，如果多次地运行程序，并且每一次都将你的程序实例设置到不同的端口上，那么操作系统应该就可以在不同的 CPU 处理器上运行每一个 Node 进程。这个例子中使用的是 localhost，但是你也可以使用另外一个服务器，这样就做到了在多个服务器上对程序进行群集。

相比这个技巧，下面一个用于服务器扩展的技巧是使用 Node 内置的特性来管理多个 Node 程序的副本。

技巧 103 使用集群保持程序的扩展性和弹性

JavaScript 程序被认为是单线程的。它们实际上使用的是单线程取决于程序运行的平台，但是从概念上来说，它们是以单线程方式运行的。这也就是说，你不得不通过一些额外的努力去充分利用多处理器或是多核处理器来扩展应用。

这个技巧会展示核心模块 `cluster`，以及它如何和你的 Node 应用的扩展性和弹性关联起来。

问题

你想提高程序的响应时间，或者是增加它的弹性。

解决方案

使用 `cluster` 模块。

讨论

在技巧 102 中，我们曾提到过在一个代理的幕后来运行多个 Node 进程。在这一个技巧中，我们将会解释它在一个 Node 是如何工作的。无论是否使用到负载均衡的代理服务器，你都可以使用本技巧的思想。不管使用哪一种方式，最终的目标都是一致的：即充分地利用处理器资源。

如图 12.8 所示展示了一个拥有两个四核 CPU 的系统，一个 Node 程序运行在该系统上，但是只能够使用一个核。

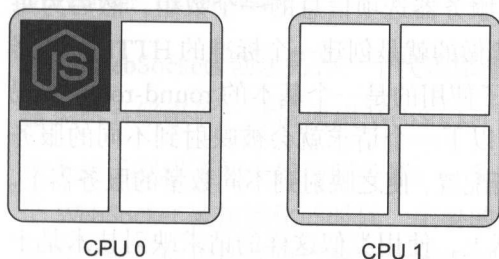


图 12.8 一个运行在单核的 Node 处理器

有一些因素造成了图 12.8 里展示的其实也不完全准确。根据相应的操作系统，进程可能围绕着处理器活动，尽管可以准确地说一个 Node 程序就是一个单进程，但是它仍然使用的是多个线程。也就是说启动一个 Express 的应用程序，它使用了一个 MySQL 的数据库、静态文件服务、用户会话等，即使它以单进程方式运行，它也仍然拥有 8 个独立的线程。

我们被强化告知 Node 程序是单线程的，主要是因为 JavaScript 的平台概念上是单线程的，但是在这背后，Node 的类库，譬如 `libuv`，就会使用线程来提供异步的 API。这就使得基于事件方式的编程风格不用过多地担心多线程的复杂度。

如果你正在部署 Node 应用，并且想让你的程序在多核处理器或者是多处理器的系统上得到更好的性能，那么你需要了解 Node 程序在这样的系统上是如何工作的。如果你正在一个多核系统上运行一个单独的应用程序，可以用类似图 12.9 的方式来表述。

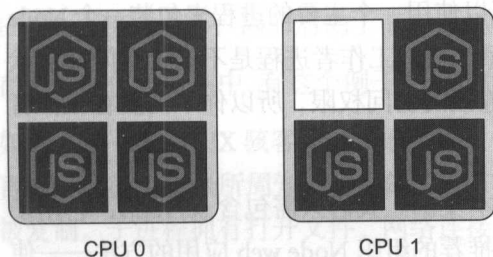


图 12.9 利用多核运行多进程

这里，我们仅在一个核上运行 Node 程序，大致的想法是，系统有一个闲置可用的核。你通过使用 `os` 核心模块得到系统的核数量，在我们的系统中，运行 `require('os').cpus().length` 返回的结果是 4——这也就是我们系统所拥有的核数量，而不是指的 CPU 的数量——Node 的 API `cpus` 方法返回的是一个数组，数组的每一个元素都描述一个 CPU。

```
[{ model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',  
  speed: 1700,  
  times:  
    { user: 11299970, nice: 0, sys: 8459650, idle: 93736040, irq: 0 } },  
{ model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',  
  speed: 1700,  
  times:  
    { user: 5410120, nice: 0, sys: 2514770, idle: 105568320, irq: 0 } },  
{ model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',  
  speed: 1700,  
  times:  
    { user: 10825170, nice: 0, sys: 6760890, idle: 95907170, irq: 0 } },  
{ model: 'Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz',  
  speed: 1700,  
  times:  
    { user: 5431950, nice: 0, sys: 2498340, idle: 105562910, irq: 0 } } ]
```

有了这个信息之后，我们可以自动地裁剪一个应用并扩展它到目标的服务器。接下来，我们需要一种方式来发叉（fork）应用程序，这样我们就可以以多进程的方式来运行它。

比如说我们运行一个 Express 的 web 程序：如何在不重写它的前提下来安全地对它进行扩展呢？主要就是通信问题：一旦你启动了一个应用程序的多个实例，你又如何访问类似数据库的共享资源呢？关于这个，有一些与平台无关的解决方案，它们需要对项目进行大量的重写——pub/sub 服务器、对象代理、分布式系统——但是我们将会使用 Node 的 cluster 模块。

cluster 模块提供了一种这样的方式，它用于运行多个工作者进程并且这些进程共享访问当前的文件句柄和套接字。这也就是说，你可以使用一个主要的进程来包装一个 Node 程序。如果你正处在一个数据库的用户访问会话当中，工作者进程是不需要访问共享状态的。所有的工作者进程都将会拥有对数据库连接的访问权限，所以你不需要在工作者进程之间启动通信。

例子 12.11 是一个使用群集的 Express 应用的基本实例。我们只需包含 server.js，它用于在 app.js 中加载主要的 Express 应用。这是我们推荐的组织 Node web 应用的方法——使用 .listen(port) 设置启动服务器的那部分在不同的文件当中而不是在程序本身里。在这个案例中，分离服务器和应用具有额外的优势，它可以使得将群集添加到项目中变得更加简单。

例子 12.11 群集一个 Node web 应用程序

```
var app = require('./app');  
var cluster = require('cluster');  
  
if (cluster.isMaster) {  
  var totalWorkers = require('os').cpus().length - 1;  
  
  console.log('Running %d total workers', totalWorkers);  
  
  for (var i = 0; i < totalWorkers; i += 1) {  
    cluster.fork();  
  }  
} else {  
  console.log('Worker PID:', process.pid);  
  app.listen(process.env.PORT || 3000);  
}
```

- ❶ 加载 cluster 核心模块。
- ❷ 决定应该启动多少个进程。
- ❸ 分叉并创建一个工作者进程。

④ 工作者进程会命中这个分支，并且 PID 会被显示。

主要的模式就是加载 cluster 核心模块①，并且然后决定有多少个核会被使用到②，如果这个是第一个（or master）进程，cluster.isMaster 允许代码运行到另外一个分支，然后 cluster.fork 按需分叉一个工作者进程③。

每一个工作者进程都会返回这个编码，所以当工作者进程命中 else 分支时，服务器就可以到工作者进程运行这个特定的代码④。在这个例子中，工作者进程启动了一个 HTTP 连接的监听，因而启动了 Express 应用程序。

在本书的代码实例中，有这个例子的完整代码，可以在 production/inky-cluster 目录下找到。

如果你是一个 UNIX 骇客，上面的代码对你来说应该会相当熟悉。fork() 的语义对 C 语言的程序员也是众所周知的。它的工作方式，是每当系统调用 fork()，当前的进程就会被复制。子进程拥有打开文件、网络连接以及内存中的数据结构的权限。为了避免性能问题，一个所谓的 copy on write 系统被使用了。这使得直到有一个写入尝试，相同的内存地址才被使用，在这点上，每一个分叉的进程都会受到一个原始的复制。当进程被分叉之后，它们就被隔离了。

还有一些额外的步骤用于处理群集的应用程序：工作进程退出复活。如果你的一个工作者进程遇到了一个问题而终止，那么你将会希望重启它。最好的事情就是，另外一个活动的工作者进程还可以继续提供服务，所以群集不仅提高了请求的延迟，同时也提高了运行时间。下面的例子就是对例子 12.11 的一个修改，用于恢复一个退出的工作者进程。

例子 12.12 从一个突然终止的工作者进程中恢复

```
var app = require('./app');
var cluster = require('cluster');

if (cluster.isMaster) {
  var totalWorkers = require('os').cpus().length - 1;

  console.log('Running %d total workers', totalWorkers);

  for (var i = 0; i < totalWorkers; i += 1) {
    cluster.fork();
  }

  cluster.on('exit', function(worker) {
    console.log('Worker %d died', worker.id);
    cluster.fork();
  });
}
```

①

②


```
} else {  
  console.log('Worker PID:', process.pid);  
  app.listen(process.env.PORT || 3000);  
}
```

❶ cluster 模块是基于事件的。

❷ 在工作进程终止之后再进行分叉。

cluster 模块是基于时间的，所以 master 可以监听像 exit 一样的事件❶，这里指代的就是工作者进程终止。这个事件的回调获得一个 worker 对象，所以你可以得到一定数量关于这个工作者进程的信息。接下来你所需要做的就只有再次 fork ❷，并且你将会得到一个全新的工作者进程进行补充。

主进程中从崩溃中恢复

你也许正在想如果当主进程终止了又会发生什么呢。为了避免这种情况的发生，即使主进程保持足够简单，也无法阻止这种可能性的发生。为了尽可能减少故障时间，你应该使用一个如 forever 模块或者 Upstart 的进程管理程序来管理集群程序。这两种程序在技巧 99 中都已经提及了。

你可以使用 Express 应用程序来运行这个实例，并且使用 kill 来强行退出工作者进程。这个会话的运行过程记录如下所示：

```
Running 3 total workers
```

```
Worker PID: 58733
```

```
Worker PID: 58732
```

```
Worker PID: 58734
```

```
Worker 1 died
```

```
Worker PID: 58737
```

上面的 3 个工作者进程一直会保持工作，直到 kill 58734 被执行，然后一个新的工作者进程就会被分叉出来，然后进程 58737 就启动了。

一旦你的群集启动了，另外要做的一件事情就是：调校（benchmark），我们将会使用 ab（<http://httpd.apache.org/docs/2.0/programs/ab.html>），一个 Apache 的调校工具，使用方式如下：

```
ab -n 10000 -c 100 http://localhost:3000/
```

它通过一次运行 100 个并行的请求来创建 10000 个请求。3 个工作者进程在我们的系统上以每秒 260 个请求的方式运行，然而一个单进程的版本只能做到每秒 171 个请求。群

集显然更快，但是它是否也在我们的前面讲到的 HAProxy 或 nginx 的 round-robin 例子中生效呢？

cluster 模块的优势就是你可以使用 Node 来编写脚本，这也就需要开发人员必须能够理解它而不是被迫去学习 HAProxy 或者 nginx 是如何做到负载均衡的。拥有额外代理服务器的负载均衡并没有像 cluster 拥有的某种内部通信的选项——你可以使用 `process.send` 和 `cluster.workers[id].on('message', fn)` 在工作者进程之间进行通信。

但是拥有专用负载均衡的代理拥有大量的均衡算法的选择，这样一来，投入一些时间测试 HAProxy、nginx 和 Node 的集群模块来发现哪一个最适合你的应用和团队是十分明智的。

专用负载均衡服务器可以将请求代理到多台服务器——你可以从一台中心服务器代理到多台 Node 应用服务器，一台服务器都是用 cluster 的核心模块来充分利用服务器的多核 CPU。

有了这样的异构设置，你将需要跟踪正在工作的应用程序，下面一节将会讲述维护生产环境下的 Node 程序。

12.3 维护

无论你的服务架构多么坚固，你也将必须面对维护你的生产系统。本节的技巧主要介绍的就是维护 Node 程序的技术，首先我们讲的是使用 npm 包的优化技术。

技巧 104 包的优化

这个技巧主要讨论 npm 以及如何使用它来实现更有效的部署，如果你觉得模块文件夹大小有点大，那么你可以通过学习本技巧的知识来解决这个问题。

问题

你将要发布到生产环境的应用程序比期望的程序大小要大得多。

解决方案

尝试使用一些 npm 的维护功能，比如 `npm prune` 和 `npm shrinkwrap`。

讨论

Heroku 要求在部署的时候你的应用程序的大小要很明确：每一次在发布的时候都会显示一个以兆字节计量的 *slug size*，以及一个在 Heroku 上最大的大小是 300MB。Slug 大小

和应用的依赖很有关系，因为程序会随着程序大小依赖的增加而不断地增加，最后你会发现这个增加是十分明显的。

即使你不使用 Heroku，你也应该注意应用程序的大小。它将会影响你部署新代码的速度，而部署新代码我们希望是尽可能快的。如果你的部署速度快，那么发布的时候遇到的缺陷就容易解决，以及新功能加入的风险也比较低。

当你通过搜查 `package.json` 里面的依赖的时候，可以清理掉一些不必要的依赖，可以通过其他一些技巧方法来减少程序的大小。`npm prune` 命令用于移除那些不在 `package.json` 里的文件包，但是它也适用于依赖自己，所以它可以明显地减少存储空间。

你应该考虑使用 `npm prune --production` 来从生产发布中移除 `devDependencies`。我们发现在生产发布中并不需要测试框架的部分。如果你将 `./node_modules` 签入到 `git` 中，那么 Heroku 将会为你运行 `npm prune`，但是它目前不运行 `npm prune --production`。

为什么要签入 `./node_modules`?

也许你会尝试将 `./node_modules` 添加到 `.gitignore` 文件中，但是千万别这么干！当你在一个将会部署的程序上工作时，那么你应该让 `./node_modules` 保存到你的代码库中。因为这可以有助于其他的人员运行你的应用程序，并且可以使得生产环境上的测试和设置与本地的一致。

当然，当你通过 `npm` 发布模块的时候，就不要这么做了，因为开源的类库应该是在安装过程中用 `npm` 来管理依赖的。

另外一个你可能会用来提高部署质量的命令是 `npm shrinkwrap`，它会创建一个名为 `npm-shrinkwrap.json` 的文件，这个文件制定了每一个依赖的明确的版本信息，而且不光是这个，它还可以递归地捕获每一个子模块的版本信息。`npm-shrinkwrap.json` 文件可以嵌入到你的代码库中。并且你也可以在部署的过程中得到每一个包的版本信息。

`shrinkwrap` 在团队合作中也是很有用的，因为这就意味着其他组员可以在开发过程中复制你机器上的模块，这有助于别人参与到一个你已经单独工作了一段时间的项目中来。

一些 PaaS 提供商还具备在部署的时候排除文件的特性。比如说，Heroku 可以接收一个 `.slugignore` 文件，它就像 `.gitignore` 一样——你可以创建一个这样的文件来忽略一些测试和一些本地的种子数据。

```
/test  
/seed-data  
/docs
```

通过充分利用 `npm` 的内置功能，你可以创建稳定且可维护性的包，减少部署的时间，并且提高部署的可靠性。

即使你的程序是易配置的、可扩展的程序，即便小心翼翼地部署了它，也还是难免会遇到问题。当出现问题的时候，我们需要有日志。继续阅读下面的技巧，我们会讨论如何处理日志文件以及日志服务。

技巧 105 日志和日志服务

如果遇到问题的時候——不能如果——应该是说当遇到问题的時候，你将需要查阅日志来发现到底发生了什么。在一个典型的服务器上，日志一般是文本类型的，但是 PaaS 提供商的日志又会是怎么样呢，比如 Heroku 和 Nodejitsu？对于这样的一些平台，它们用的是日志服务。

问题

你想在自己的服务器上（或者是 PaaS 提供商）运行的 Node 程序中记录日志。

解决方案

要么使用 `logrotate` 将日志记录到文件中，要么使用第三方日志服务。

讨论

在 UNIX 中，一切东西都是文件，并且这也是系统管理人员和运维专家对日志的通常的认识方式。日志只不过就是文件：通过将程序的流数据写入到文件，再将流数据从文件读出。这对于我们那些喜欢使用命令行工具的人来说是十分方便的——通过使用 `grep`、`sed` 和 `awk` 这样的命令来传输（pipe）文件，即使是对于千兆字节的文件来说也是一件很轻松的事情。

因此，无论你怎么做，你都会想正确地使用 `console.log` 和 `console.error`。你还有必要知道 `err.stack`——它是 Node 中 `Error` 的一个实例，它在定义的时候会拿到一个 `stack` 的属性，这个对于在生产环境下调试和解决问题是十分有帮助的。对于更多的关于写日志的知识，你可以参考第 2 章中的技巧 6。

使用 `console.error` 和 `console.log` 的好处是你可以将内容数据传输到不同的地方。下面的命令会将数据从一个标准的输出（`console.log`）重定向到一个 `application.log`，将一个标准的异常 `console.error` 重定向到一个 `errors.log`：

```
npm start 1> application.log 2> errors.log
```

这里你需要记住的是，大于号表示重定向输出，并且使用一个特定的数字来标识输出流：1 是标准输出，2 是标准异常。

不用多久，你的异常文件就会变得特别巨大。幸运的是，现代的 UNIX 系统通常都有日志轮转包，它用于将文件通过时间片进行分割并且对其进行压缩。可以在 Debian 或者是 Ubuntu 上通过使用 `apt-get install logrotate` 安装 `logrotate` 包，一旦你安装好了它，你将需要为每一组要创建的日志文件用到一个配置文件。下面展示了一个配置的例子，你可以通过一定的修改来适应应用程序。

例子 12.13 logrotate 配置

```
"/var/www/nodeapp/logs/application.log"
"/var/www/nodeapp/logs/application.err" {
    daily
    rotate 20
    compress
    copytruncate
}
```

❶
❷
❸
❹

- ❶ 每一天都运行。
- ❷ 保持 20 个文件。
- ❸ 压缩轮转文件。
- ❹ 截取当前的日志文件。

通过罗列出你需要轮转的日志文件，可以列出一些你想要用到的选项。`logrotate` 有很多选项，并且你可以通过使用 `man logrotate` 查阅具体的选项。我们这里第一个要用到的选项就是 `daily` ❶，它指示的意思是我们每天都要轮转日志文件。下一行选项设置说的是使得 `logrotate` 保持 20 个文件。在文件被移除之后❷，第三个选项将会确保老的日志文件会被压缩，这样不会消耗太多的空间❸。

第四个选项 `copytruncate` ❹对于一个使用基于简单标准 I/O 日志的程序是非常重要的，它使得 `logrotate` 拷贝并截断当前的日志文件。这就意味着你的应用程序不需要关闭或者重新打开标准输出端——它即使没有特殊的配置应该也是可以工作的。

在一台服务器上运行的简单应用程序使用标准 I/O 和 `logrotate` 肯定是没有问题的，但是如果在一个集群上运行一个应用程序，管理日志就不会那么简单了。有一些专门致力于在集群环境下提供日志服务的模块，有些人甚至非常喜欢这样的模块，因为这些模块在标准的文件格式中生成输出。

`log4node` (<https://github.com/bpaquet/log4node>) 的使用方式和 `console.log` 很相似，但是它有些特性使得它更加容易在集群环境下使用。它为所有的工作者进程创建一个日志文件，并且通过监听一个 `USR2` 信号来决定何时重新打开文件。它支持配置选项，这些

选项包括日志的级别、日志的前缀，所以你可以保持日志在生产环境的测试期间安静地增加。

winston (<https://github.com/flatiron/winston>) 是一个支持多传输的日志模块，包括 Cassandra，它允许以集群方式写日志。也就是说，如果你有一个程序在一个小时之内需要写入百万条日志信息，那么你可以以一种更加可靠的方式使用多个服务器来捕获日志。

winston 支持远程的日志服务，包括 Papertrail 这样的商业服务。Papertrail 和 Loggly (查看图 12.10) 都是可以帮你输送日志的商业服务，通常使用的是 syslogd 协议进行传输。它们也会对日志进行索引，所以即便是搜索千兆级别的日志也相当快，这取决于你的查询。

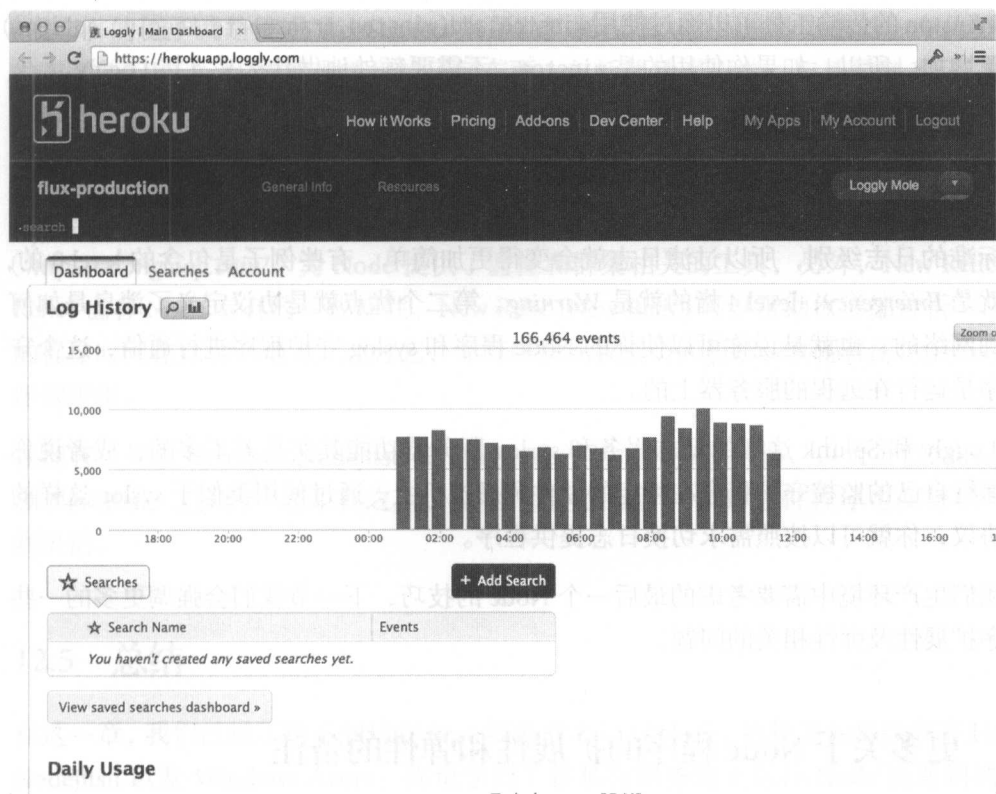


图 12.10 Logglyd 面板界面

一个类似于 Loggly 的服务对于 Heroku 来说是相当重要的，Heroku 只存储了最近的 5000 条日志，通过运行一个典型的应用可以在短时间内冲掉其他的日志。如果你将 Node 应

用程序部署到 Heroku, 它使用的是 `console.log`、`log4node` 或者 `winston`, 那么你将可以通过启动这些插件对你的日志进行重定向操作。

在 Heroku 中, Loggly 是可以通过选择一个计划名字并且在你的项目目录下运行 `heroku addons:add Loggly:PlanName` 来进行配置的。输入 `heroku addons:open loggly` 将会打开 Loggly 的 web 界面, 但是你可以直接通过在 *Resources* 下面的管理面板下单击链接打开。任何一个标准的 I/O 日志都会直接发送到 Loggly。

如果你使用的正是 `winston`, 那么也有 Loggly 可用的传输工具, 其中一个是 `winston-loggly` (<https://github.com/indexzero/winston-loggly>), 它可以在你私有服务器上简单地访问 Loggly, 使用的是非 Heroku 服务。

因为 `Winston` 的传输工具可以通过使用 `winston.add(winston.transports.Loggly, options)` 来配置增加, 所以, 如果你使用的是 `winston`, 不需要额外地做什么来支持 Loggly。

在你的程序中有一个用于日志的标准, 那就是 `Syslog` 协议。Syslog 消息包有一个标准的数据格式, 所以你不需要手动地生成日志格式。使用 `winston` 这样的模块, 一般都支持 `syslog`, 所以你可以在 Node 程序中使用它, 主要有两个优点: 第一个优点就是日志消息具有标准的日志级别, 所以过滤日志就会变得更加简单。有些例子是包含的 `level 0` 的, 指的就是 *Emergency*, `level 4` 指的就是 *Warning*。第二个优点就是协议定义了消息是如何发送到网络的, 也就是说你可以使你的 Node 程序和 `syslog` 守护程序进行通信, 这个守护程序是运行在远程的服务器上的。

类似 Loggly 和 Splunk 这样的日志服务和 `syslog` 服务器功能其实是差不多的, 或者说你可以运行自己的监控守护程序在特定的硬件和虚拟机上, 通过使用类似于 `syslog` 这样的标准协议, 你就可以按照需求切换日志提供程序。

这是我们生产环境中需要考虑的最后一个 Node 的技巧, 下一节我们会强调更多的一些和程序扩展性及弹性相关的问题。

12.4 更多关于 Node 程序的扩展性和弹性的备注

在这一章, 我们已经展示了如何使用代理和群集模块来扩展 Node 程序。我们说到的一个集群的好处就是进程间通信更加容易。在独立的服务器上运行一个应用程序, 如何能够使得进程之间进行通信呢?

这个问题的简单的回答可能是 HTTP——你可以构建一个内部的 REST API 来进行通信。如果你的消息需要更快的响应, 你甚至可以使用 `WebSockets`。当遇到这个问题的时候,

我们使用了 RabbitMQ (<https://www.rabbitmq.com/>), 这允许每一个 Node 应用通过共享消息总线相互发消息进行通信, 从而做到通过集群进行分布式地工作。

搜索引擎项目使用的是 Node 程序进行下载和收录内容。它的主要工作分为蜘蛛爬程序、下载和收录。大批的 Node 进程都会从队列中获取工作任务, 同时也将新工作再放回队列中。

NPM 的 RabbitMQ 客户端可以有多种实现方式——我们使用的是 amqplib (<https://www.npmjs.org/package/amqplib>), 也有一个和 RabbitMQ 旗鼓相当的工具 zeromq (<http://zeromq.org/>), 它是一个高度专注和值得选择的工具。

另外一个选项是使用一个托管的 publish/subscribe (发布/订阅) 的服务。一个 Publish (发布) 的例子就是使用 WebSockets 来帮助扩展程序, 使用这种方法的一个优势就是, 发布者 (Publisher) 能够发消息给任何东西, 包括移动客户端。与其限制 Node 程序进行消息传递, 还不如创建一个消息通道, web、移动设备, 甚至是桌面客户端都能够通过这个消息通道订阅消息。

最后, 如果你正在使用私人服务器, 那么你将会需要监控你的资源使用情况。StrongLoop (<http://strongloop.com/>) 为 Node 提供了监控和群集相关的工具, 另外, New Relic (New Relic) 也有 Node 专有的一些特性。New Relic (New Relic) 可以帮助你详细分析一个运行的程序时间花在什么地方, 通过它, 你可以发现数据访问瓶颈, 发现视图的渲染以及程序的逻辑。

在拥有了类似 Heroku、Nodejitsu 和 Microsoft 这样的服务提供商, 以及类似 StrongLoop 和 New Relic 这样的一些工具之后, 在生产环境下运行 Node 软件程序也变得相当成熟和灵活。

12.5 总结

在这一章, 我们已经了解了如何在 PaaS 提供商上运行 Node, 所提及的提供商有 Heroku、Nodejitsu 以及 Windows Azure。你也学到了在私有服务器上运行 Node 会遇到的问题: 如何安全地访问 80 端口 (技巧 98), 以及 WebSockets 是如何关联到生产需求的 (技巧 100)。

不管你的代码运行有多快, 如果你的应用程序是受大众欢迎的, 那么你可能就会遇到性能问题, 我们在这部分学习了缓存 (技巧 101)、代理 (技巧 102), 以及使用集群来扩展程序 (技巧 103) 的内容。

为了保持程序能够坚固运行,我们也介绍了生产环境中的 npm 上维护相关的技术(技巧 104)以及日志(技巧 105)。如果你遇到了一些问题,那么相信你应该已经掌握了足够的信息去解决它了。

接下来,你应该要调节如何构建 Node web 应用程序,并且在一种可维护和可扩展的状态下发布程序。

第三部分

编写模块

我们已经了解了 Node 的核心库，并且掌握了实践中的一些技巧，我们一直在讲述关于 Node 生态系统最大一部分：即通过第三方模块开发的社区驱动创新。核心提供了我们构建需要的基础，实践提供了自信创造的工具和洞察力，最终可以创造出什么都取决于我们自己！

最后一部分的内容会带你了解如何构建一个模块并且回馈给社区全部的里里外外的东西。

13

编写模块，掌握 Node 的所有

本章概要

- 计划开发一个模块
- 设置 `package.json`
- 依赖处理和语义化版本号
- 添加可执行脚本
- 模块测试
- 发布模块

Node 包管理工具（`npm`）可以说是至今为止在多个平台中极好的一个包管理。`npm` 核心是提供安装、管理、创建 Node 模块的工具集合。使用 `npm` 的门槛很低，而且是那么整洁和方便，一切就是刚刚好。如果你对此还没有信心，那么希望这一章的内容可以帮助你的一切做得更好。

这一章的副标题是“掌握 Node 的所有”。之所以会有这么一个标题是因为我们觉得 Node 用户构建的模块正是 Node 生态系统中最重要的一部分。Node 核心开发团队在早期就决定，Node 应该有一个精简的标准库，只包含足够强大的核心功能，用于构建足够强大的模块。我们明白这个核心是极其重要的建筑根基，所以我们将创建模块这一章内容放到了最后。在 Node 中，你可以发现某个特定的协议或者客户端会有 5 到 10 个不同的模块实现。我们觉得这个很棒，因为可以借由实验来推动创新。

在实践中可以发现，更小的模块更易于维护。大型的模块会变得异常难以维护和测试。Node 允许多个小模块很简单地组合到一起来解决愈来愈复杂的问题。

Node 的 `require` 系统（基于 CommonJS 规范：<http://wiki.commonjs.org/wiki/Modules/1.1>）用于有序地管理依赖，来防止依赖地狱。它可以很轻松地处理依赖同一个模块多个版本的问题，如图 13.1 所示。

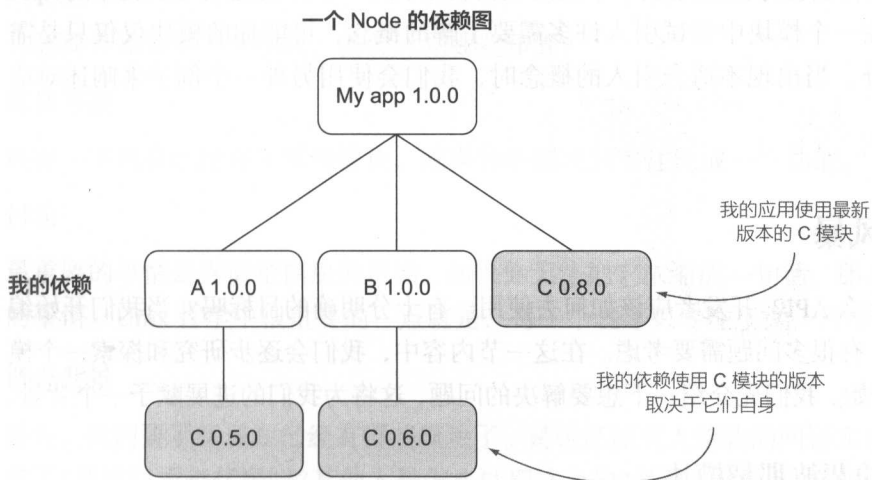


图 13.1 Node 防止依赖地狱

除了标准的模块依赖，你还可以指定开发时使用的依赖和同等的依赖（稍后会详述），`npm` 会帮助处理好相关问题。

依赖图

如果你希望查看项目的依赖图，可以在项目根目录使用 `npm ls` 来获取。

另外一点不同的是在 `npm` 早期历史中决定和 Ruby 的 `bundler` `gem` 一样流行地在默认的本地层级下管理依赖。这会在你的项目内部处理模块（在 `node_modules` 目录下），因为没有全局共享的模块状态，会让跨多个项目使用时出现依赖地狱。

安装全局模块

如果需要，你可以使用 `npm install -g module-name` 来安装全局的模块，这在你需要一个全局级别可运行的模块是很有用的。

希望我们可以激起你探索模块开发的欲望，这一章中我们将围绕下边相关的技术点展开：

- 尽可能发挥 `package.json` 的作用。
- 利用 `npm` 来完成多种模块开发的任务。
- 模块开发的最佳实践。

我们讲解的内容将会循序渐进地从一个空的项目目录，到一个开发完成可以发布的 `npm` 模块。我们会在一个模块中尝试引入许多需要了解的概念，可能你的模块仅仅只是需要其中的一部分。当出现不适合引入的概念时，我们会使用另外一个例子来阐述对应的内容。

13.1 头脑风暴

我们想要创建什么 API？开发者应该如何去使用。有十分明确的目标吗？当我们开始编写一个模块前，有很多问题需要考虑。在这一节内容中，我们会逐步研究和探索一个模块的想法。一开始，我们先介绍一下想要解决的问题，这将为我们的进展赋予一个背景。

13.1.1 更快的斐波那契模块

在 Node 历史中，有一个相当著名的对 Node 的批判（虽然可以说是误导），名为“Node.js is Cancer”（<http://pages.citebrite.com/b2x0j8q1megb>），其中作者提到在 Node 单线程的系统中，运行一个 web 服务器，并且执行 CPU 密集型的任务，表现将会异常糟糕。

这个实现是一个很常见的斐波那契数据的递归运算（http://en.wikipedia.org/wiki/Fibonacci_number），可以很简单实现，如下：

```
function fibonacci (n) {  
  if (n === 0) return 0;  
  if (n === 1) return 1;  
  return fibonacci(n-1) + fibonacci(n-2);  
}
```

❶

❶ 这一行是新增的，因为原始的实现并没有给数列返回正确的 0。

以上的代码实现在 V8 引擎中是很慢的，在 JavaScript 中并没有实现尾递归优化，由于堆栈溢出而无法进行非常高的数据运算。

我们可以编写一个模块来帮助解决斐波那契计算缓慢的问题，并借此来了解模块从开始到结束的整个开发流程。

技巧 106 计划编写我们的模块

当我们要开始编写一个模块时，我们应该怎么办？在我们开始编写第一行代码之前有什么可以做的？事实证明，提前规划是非常有用的，可以帮助我们在一路上减少痛苦。我们来看一下如何把它做好。

问题

你想要开始编写一个模块，需要计划哪些步骤？

解决方案

调查一下现在已经有了哪些模块，确保你的模块只专注完成一个功能。

讨论

最重要的事情是弄清楚模块的目的。如果你无法把它浓缩成一句话，那么可能会做过多的事情。Unix 哲学中很重要的一点就是，每一个程序只专注于做一件事情。

调查背景

首先，我们要了解现在已经有有哪些模块了。是否已经有人给我的问题实现了一个解决方案了？我可以贡献代码吗？其他人是怎么处理这个的？最简单的调查方法就是在 npmjs.org 中搜索或者使用以下的命令：

```
npm search fibonacci
```

❶

❶ 如果是第一次运行 `npm search`，在你获得结果前需要等待一段时间来更新本地的缓存。

我们看一下一些有趣的结果：

```
fibonacci Calculates fibonacci numbers for one or endless iterations,...  
=franklin 2013-05-01 1.2.3 fibonacci math bignum endless
```

```
fibonacci-async So, you want to benchmark node.js with fibonacci once...  
=gottox 2012-10-29 0.0.2
```

```
fibonacci-native A C++ addon to compute the nth fibonacci number.  
=avianflu 2012-03-21 0.0.0
```

这里我们可以看到不同实现模块的名称和描述。还可以看到哪个版本在什么时间发布的。看起来有两个是比较旧的，并且版本号比较低，这意味着 API 还在持续调整中。第一个版本在 1.2.3 的结果看起来比较稳定，近来也有更新。可以使用以下命令来获取更多的相关信息：

```
npm docs fibonacci
```

`npm docs` 这个命令可以加载模块的首页（如果有），或者 `npmjs` 的搜索结果，我们可以参考图 13.2。

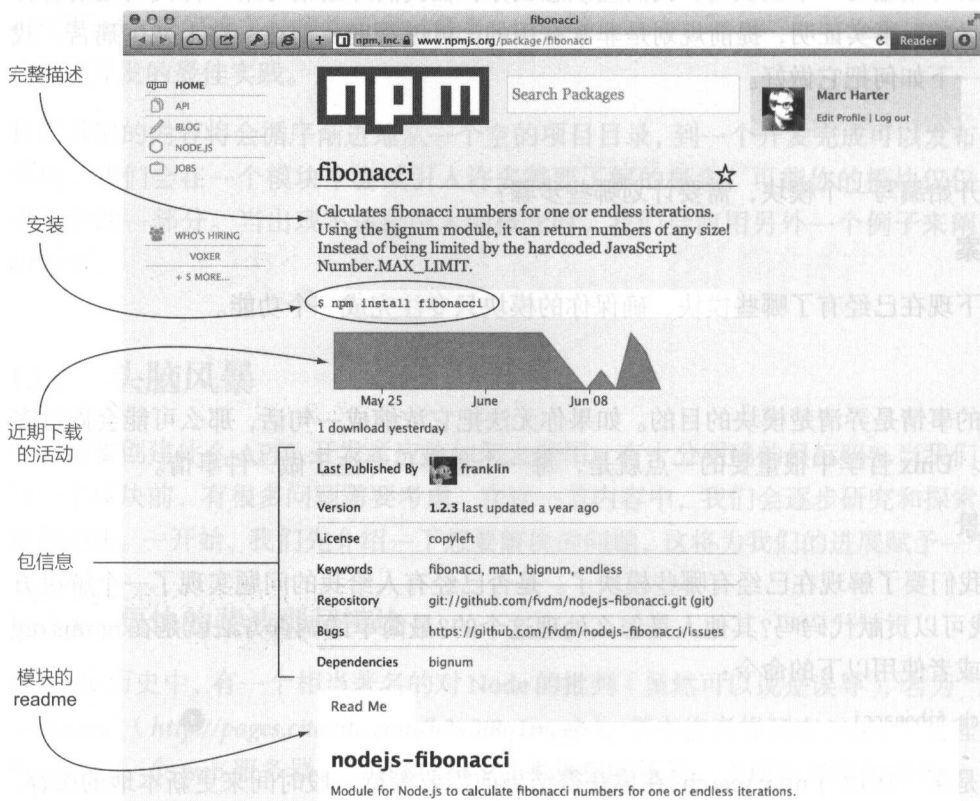


图 13.2 npmjs.com 模块详情页面

`npmjs` 结果页面可以对模块的总体情况有个大概的了解。我们可以看到这个模块依赖于 `bignum` 这个模块以及一年前更新过，还可以看到它的 `readme` 文档来获取 API 的信息。

虽然这个模块看起来还不错，但是我们还是来创建一个模块做一个实验，可以尝试实现一些其他的想法来处理斐波那契数列。在例子中，我们会创建一个不依赖于 `bignum` 的模块，尝试使用不同的实现方式，直接使用 JavaScript 来测试模块的性能。

只做好一件事情

一个模块应该是简单的、方便使用的。我们尝试使用一句话来描述模块的目的：

使用 JavaScript 尽可能快地来计算斐波那契数列。

这是一个非常好的开始：明确和简洁。当这个概念无法满足你的要求时，那便是我们超出了界限，便需要编写另外一个模块来进行扩展了，而不是在原本的基础上再添加功能。对于这个项目，如果要添加一个 web 服务器来返回函数的运行结果，最好的方式是编写一个新的模块依赖于现有的这个模块来处理。

当然，这并非硬性的规定。但它有助于我们为目标用户弄清楚模块的功能和目的。相关的描述可以添加到我们模块的 `package.json`（相关内容稍后会详述）文件中，以及 `readme` 文档中。

我们需要一个模块名称，一开始这并不是很重要的，但是为了在接下来的例子中使用，把模块命名为 `fastfib`。创建一个 `fastfib` 的目录来开始我们的模块：

```
mkdir fastfib && cd fastfib
```

现在已经定义了我们的模块需要去做的“一件事”，也创建对应的目录，我们会在下一节的内容验证我们的想法，看看它是否能够成功。

技巧 107 验证我们模块的想法

我们关注接下来要做的事情，要证明我们的想法是可行的。这也是考虑模块的 API 层面的时候。它是否可用？是否达到了我们的目的？下面我们一起来看看吧。

问题

我们应该先编写什么代码来验证自己的想法是否可行？

解决方案

使用 TDD 来检查 API 层面的使用。

讨论

了解想要实现的模块是如何发挥作用的，这一点很重要。在 `fastfib` 中，我们同步计算一个斐波那契数列的结果。我们能够想到的最简单、最方便的 API 是怎么样的？

```
fastfib(3) // => 2
```

对，就像一个简单的函数调用，然后返回结果。

当构建一个异步 API 时，建议使用 Node 标准的回调方式，它可以和很多异步控制流的类库很好地协同工作。如果我们的模块是异步的，那么看起来就像这样：

```
fastfib(3, function (err, result) {  
  console.log(result); // => 2  
});
```

我们已经有了一个异步的 API。在这一章开始的时候，已经展现了一个希望能够提升的代码实现。所以我们有了一个对比的基础来评估其他的实现代码。在项目中创建一个 lib 目录和一个 recurse.js 文件来存放递归的实现代码：

```
module.exports = recurse;
```

```
function recurse (n) {  
  if (n === 0) return 0;  
  if (n === 1) return 1;  
  return recurse(n-1) + recurse(n-2);  
}
```

❶ 对外暴露一个简单的函数来匹配我们的 API 设计。

定义入口

每一个模块都有一个程序入口：它可以是一个对象、函数、构造函数，当我们任何时候使用 require 来引入时便可以获取到。我们明确在模块中会尝试使用多种不同的实现方法，不希望使用 lib/recurse.js 来作为入口，因为入口可能会改变。

通常在项目根目录中的 index.js 都会被作为入口。很多时候入口会自然而然地做到最小，仅仅只是把提供 API 的多个部分组合在一起提供给用户。我们现在创建一个文件，内容如下：

```
module.exports = require('./lib/recurse');
```

现在当用户使用 require('fastfib') 时便会引用这个文件，来获取对应的递归实现。当我们想要调整 API 的具体实现时，只需要修改这个文件便可以。

测试实现代码

现在我们有 fastfib 的第一个实现。我们来确保以下这个实现是可行的。为此，创建一个 test 的目录，在里边使用一个 index.js 文件来编写测试：

```
var assert = require('assert');  
var fastfib = require ('../');
```

```
assert.equal(fastfib(0), 0);  
assert.equal(fastfib(1), 1);  
assert.equal(fastfib(2), 1);  
assert.equal(fastfib(3), 2);  
assert.equal(fastfib(4), 3);  
assert.equal(fastfib(5), 5);  
assert.equal(fastfib(6), 8);  
assert.equal(fastfib(7), 13);
```



```
assert.equal(fastfib(8), 21);
assert.equal(fastfib(9), 34);
assert.equal(fastfib(10), 55);
assert.equal(fastfib(11), 89);
assert.equal(fastfib(12), 144);
```

// 如果可以运行到这里，我们则假定是正确的

我们来运行一下测试用例：

```
node test
```

看不到任何错误抛出，那么至少可以说明这个实现在目前还是可行的。

基准测试

我们已经有了定义好的 API 和对于模块的简单测试，如何确定它的性能如何，跑得快不快？为此我们可以使用 jsperf.com 项目中的 JavaScript 的基准测试工具 Benchmark.js (<http://benchmarkjs.com/>)。我们在项目中引用它：

```
npm install benchmark
```

我们创建一个 benchmark 的目录，添加一个 index.js 文件来编写基准测试代码：

```
var assert = require('assert');
var recurse = require('../lib/recurse');
var suite = new (require('benchmark')).Suite;

suite
  .add('recurse', function () { recurse(20); })
  .on('complete', function () {
    console.log('results: ');
    this.forEach(function (result) {
      console.log(result.name, result.count, result.times.elapsed);
    });
    assert.equal(
      this.filter('fastest').pluck('name')[0],
      'recurse',
      'expect recurse to be the fastest'
    );
  })
  .run();
```

❶ 引入我们的递归实现。

❷ 创建一个新的基准用例。

- ❸ 为递归的实现方式添加一个测试，计算斐波那契数列的 20 个数字。
- ❹ 在测试完成后统计结果。
- ❺ 打印测试名称、时间量，以及在特定时间量中跑的迭代次数。
- ❻ 断言递归的实现方式是最快的。当前只有一个实现方式，这应该是很简单的。

我们可以在根目录下跑一下基准测试：

```
$ node benchmark
results:
recurse 392 5.491
```

看起来我们可以在大约 5.5 秒的时间内计算 `recurse(20)` 392 次。我们看一下还能否再提升一下。原来的递归实现并没有尾递归的优化。所以我们可以在这一点上做优化处理。给 `lib` 目录添加多一个 `tail.js` 文件来实现：

```
module.exports = tail;

function tail (n) { return fib(n, 0, 1); }
function fib (n, current, next) {
  if (n === 0) return current;
  return fib(n - 1, next, current + next);
}
```

❶
❷
❸
❹

- ❶ 为了和原来的递归实现代码提供相同的 API，我们添加这个方法设置默认的值。
- ❷ 这个递归方法需要传递下一个索引值 `n`、数列中当前的数字，以及数列中下一个数字。
- ❸ 数列最开始的时候，返回数列当前的数字。
- ❹ 运行下一个调用。这是在尾部的位置，因为计算在递归函数调用之前就开始了，因此能够通过编译器来进行优化。

现在把测试添加到 `benchmark/index.js` 文件中，看一下这个实现是否能够做得更好：

```
var recurse = require('../lib/recurse');
var tail = require('../lib/tail');
---
.add('recurse', function () { recurse(20); })
.add('tail', function () { tail(20); })
```

❶
❷

- ❶ 引用尾递归优化的实现代码。
- ❷ 在原有的递归实现测试之后添加尾递归的测试。

我们来看一下结果：

```
$ node benchmark
results:
recurse 391 5.501
tail 269702 5.469
```

❶

```
assert.js:92
```

```
  throw new assert.AssertionError({
    ^
```

```
AssertionError: expect recurse to be the fastest
```

❷

❶ 把递归放到函数尾部位置运行速度有 689 倍的提升。

❷ 我们的断言失败了，递归的实现不再是最快的。

哇！尾递归真的帮助我们斐波那契的计算速度提升了不少。所以我们把它作为默认的实现方式，在 `index.js` 文件中进行切换：

```
module.exports = require('lib/tail');
```

确保测试可以通过：

```
node test
```

没有错误，这个看起来仍然很棒。如之前提到的，适当的尾递归调用有可能导致堆栈太大，而在 JavaScript 中并不支持过大的堆栈。所以我们尝试另外一种实现方式，看一下还能否做得更好。为了避免过大的数列导致堆栈溢出，使用迭代的方式来实现，在 `lib/iter.js` 文件中编写：

```
module.exports = iter;
```

```
function iter (n) {
```

```
  var current = 0, next;
```

```
  for (var i = 0; i < n; i++) {
```

```
    swap = current, current = next;
```

```
    next = swap + next;
```

```
  }
```

```
  return current;
```

```
}
```

❶

❷

❸

❶ 设置当前值。

❷ 进行 n 次的迭代，每一次交换数列当前值和数列中的下一个值，再进行相加作为下一个值。

❸ 返回最后的结果。

我们把这个实现添加到 `benchmark/index.js` 的测试中：

```
var tail = require('../lib/tail');
var iter = require('../lib/iter');
---
.add('tail', function () { tail(20) })
.add('iter', function () { iter(20) })
```

❶

❷

❶ 在尾递归的实现代码引用后边引入迭代的实现。

❷ 在尾递归的测试后添加迭代方式的测试。

我们看一下结果：

```
$ node benchmark
results:
recurse 392 5.456
tail 266836 5.455
iter 1109532 5.474
```

使用迭代的方式比尾递归的版本还要快上 4 倍左右，而比最初的方法则是快了 2830 倍。看起来我们证明了 fastfib 的实现是可行的，更新一下 benchmark/index.js 的断言，iter 的方式应该是最快的：

```
assert.equal(
  this.filter('fastest').pluck('name')[0],
  'iter',
  'expect iter to be the fastest'
);
```

然后更新我们的入口，使用最快的版本：

```
module.exports = require('../lib/iter');
```

跑一下测试来确保实现是正确的：

```
node test
```

还是没有错误，我们是正确的！如果之后我们发现 V8 优化了尾递归的调用而导致尾递归的方式比迭代的方式还要快，那么我们的基准测试断言便会失败，只需要调整实现的方式便可。回顾一下整个项目的结构：

```
fastfib
├── benchmark
│   └── index.js
├── index.js
├── lib
│   ├── iter.js
│   ├── recurse.js
│   └── tail.js
```

❶

❷

❸

```
├── node_modules
│   └── benchmark
├── test
│   └── index.js
```

4

5

❶ 基准测试。

❷ 模块入口。

❸ 具体实现。

❹ 安装的依赖。

❺ 测试代码。

看来我们已经证明了我们的想法是可行的。最重要的是去尝试！尝试不同的实现方式！你可能不会一开始就是对的，多尝试几次直到满意。在这一节中，我们试了三个不同的实现才确定我们要的一个。

接下来去看看模块开发的下一步，创建一个 package.json 文件。

13.2 创建 package.json 文件

现在我们已经有一个有趣的想法，并且证明了是可以实现的，我们需要在 package.json 这个文件中去描述要开发的模块相应的一些配置。

技巧 108 创建 package.json 文件

package.json 是一个很重要的文件，用于管理模块的通用脚本、依赖等核心的配置数据。不论你是否对模块进行发布，或者简单地管理一个内部的项目，配置好一个 package.json 文件能够帮助你更好地进行开发。在这一节内容中，我们会讲到如何配置 package.json 以及使用 npm 来帮助你设置 package。

问题

你需要创建一个 package.json 文件。

解决方案

使用内置的 npm 工具。

讨论

npm init 命令可以提供一个很方便的接口来一步步地配置 package.json。我们在 fastfib 项目的根目录下执行这个命令：

```
$ npm init
---
name: (fastfib) fastfib
version: (0.0.0) 0.1.0
description: Calculates a Fibonacci number as fast as possible
  with only JavaScript.
entry point: (index.js)
test command: node test &&
  node benchmark
git repository: git://github.com/wavded/fastfib.git
keywords: fibonacci fast
author: Marc Harter <wavded@gmail.com> (http://wavded.com)
license: (ISC) MIT
---
```

1
2
3
4
5
6
7
8
9

- ❶ 模块的名称，发布时必填项，由于我们的项目目录名称为 `fastfib`，所以默认为 `fastfib`。
- ❷ 模块的版本。这也是必填项，强烈建议使用语义化的版本号（`semver`）。我们等一下再详细讲 `semver`。我们从版本 0.1.0 开始。
- ❸ 对模块比较详细的描述。这里使用简洁的一句话定义来描述我们这个模块。
- ❹ 模块的入口文件配置，模块加载时首先加载的文件。
- ❺ 测试的执行命令。配置之后可以使用 `npm test` 来执行对应的命令。考虑到我们的模块需要保证正确和速度，所以 `node test` 和 `node benchmark` 两者都需要运行。
- ❻ 存放代码的 Git 仓库的地址。如果你已经进行 git 仓库的初始化和添加了远程 git 地址，这个会自动帮你补全。
- ❼ 搜索模块时的关键字。
- ❽ 作者的信息！使用以下格式：[名称] < 邮箱地址 > ([网站地址])。
- ❾ 这是一种软件发布许可证，默认是 ISC： http://en.wikipedia.org/wiki/ISC_license。如果不确定使用哪一个许可证，可以看一下这个网站： <http://choosealicense.com/>。

package 配置项

需要了解更多关于 package 的配置项详情，可以使用 `npm help json` 来查看离线的帮助文档（<https://www.npmjs.org/doc/json.html>）。

当配置好默认的用户信息（`$HOME/.npmrc`），运行 `npm init` 会变得更加简单，默认的值都会帮助你先设置好。这里是可配置的选项：

```
npm config set init.author.name "Marc Harter"
npm config set init.author.email "wavded@gmail.com"
npm config set init.author.url "http://wavded.com"
npm config set init.license "MIT"
```

使用了这些配置后，`npm init` 不会再询问你 `author` 相关的信息，而直接帮你补全。当然，默认也使用 MIT 的许可证。

已存在的模块

如果你在设置 `package.json` 之前已经安装了模块，`npm init` 会自动帮助你把模块以正确的版本补充到 `package.json` 文件中。

当完成初始化后，在目录中就有一个看起来很棒的 `package.json` 的配置文件了，看起来就像下边这样：

```
{
  "name": "fastfib",
  "version": "0.1.0",
  "description": "Calculates a Fibonacci number as fast
    as possible with only JavaScript.",
  "main": "index.js",
  "bin": {
    "fastfib": "index.js"
  },
  "directories": {
    "test": "test"
  },
  "dependencies": {
    "benchmark": "^1.0.0"
  },
  "devDependencies": {},
  "scripts": {
    "test": "node test && node benchmark"
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/wavded/fastfib.git"
  },
  "keywords": [
    "fibonacci",
    "fast"
  ],
  "author": "Marc Harter <wavded@gmail.com> (http://wavded.com)",
```

```
"license": "MIT",  
"bugs": {  
  "url": "https://github.com/wavded/fastfib/issues"  
},  
"homepage": "https://github.com/wavded/fastfib"  
}
```

- ❶ 模块的依赖。注意 `npm init` 会发现我们已经安装了 `benchmark` 模块，然后自动帮我们加上。
- ❷ 仅用于开发的一些依赖。某个用户安装你的模块时并不会安装这些开发依赖。
- ❸ `npm bugs` 时加载浏览器打开这个地址，给用户反馈使用的问题。当我们使用 Github 仓库，`npm` 会自动把这个配置给我们补上。
- ❹ 项目的首页地址。我们使用 Github 仓库时会自动使用 Github 的地址。执行 `npm docs` 命令时会使用浏览器打开这个地址。

我们已经有一个配置好的 `package.json` 文件了，可以直接修改 JSON 文件或者用 `npm` 一些命令来增加更多的属性或者修改某些属性。`npm init` 命令只是接触了 `package.json` 比较表面的东西。我们接下来会看一下更多一些可以添加到配置中的东西。

下一章我们会了解到关于 `package.json` 更多的东西，以及模块开发其他方面的内容。

技巧 109 依赖处理

Node 在 `npm` 上已经有 80000 多个发布的模块。在 `fastfib` 模块中，我们已经使用到了其中一个：`benchmark`。在 `package.json` 定义好依赖，在我们自己或者用户安装使用时，可以帮助维持模块的完整性。使用 `npm install` 时 `package.json` 文件会告诉 `npm` 去获取哪些正确版本的依赖模块。当获取 `package.json` 文件中声明的依赖失败时会抛出错误。

问题

我们如何有效地管理依赖。

解决方案

使用 `npm` 来保持 `package.json` 文件中依赖配置的同步。

讨论

`package.json` 允许你定义四种类型的依赖对象，如图 13.3 所示。

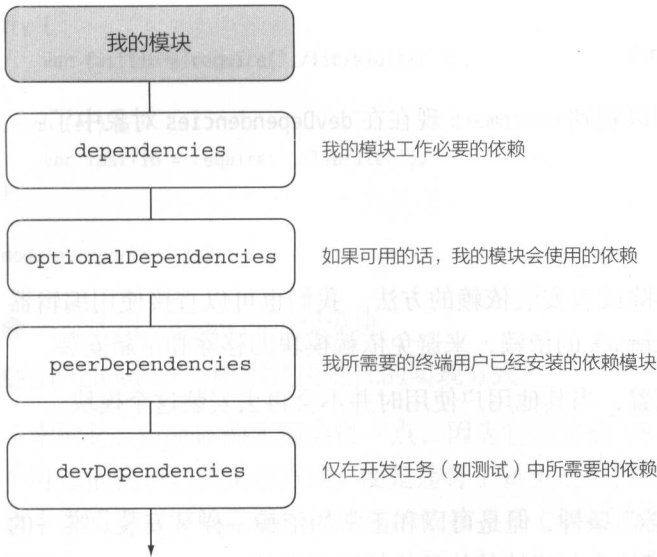


图 13.3 依赖的不同类型

以下是依赖的类型列表:

- dependencies——模块正常运行需要的依赖。
- devDependencies——开发时需要的依赖, 如测试、基准测试和服务器加载工具等。
- optionalDependencies——非必要的模块依赖, 某种程度上可以增强模块。
- peerDependencies——运行时需要的依赖, 但已经被安装了, 例如 grunt。

我们反过来看一下项目, 在 package.json 中添加或者移除一些配置。

主要依赖和开发依赖

我们已经使用 npm init 初始化了 package.json 文件, 在 dependencies 中已经有了 benchmark 的模块。看一下列表, 有一些配置并不是正确的。首先入口文件并不会引用 benchmark 模块, 所以用户并不需要它:

```
index.js requires ./lib/iter.js which requires nothing
```

基准测试仅仅只是在我们开发模块时使用到。所以把它从依赖中移除, 使用 npm remove 命令加上 --save 标志来把它从 package.json 配置文件中移除:

```
$ npm remove benchmark --save
unbuild benchmark@1.0.0
```

然后可以使用 npm install 加上 --save-dev 把它安装到开发依赖中去:

```
$ npm install benchmark --save-dev
benchmark@1.0.0 node_modules/benchmark
```

现在看一下 package.json 文件，可以看到 benchmark 现在在 devDependencies 对象中了：

```
"devDependencies": {
  "benchmark": "^1.0.0"
},
```

这便是使用 npm 的相关命令来移除或者安装依赖的方法。我们也可以直接使用编辑器修改 package.json 文件来调整 benchmark 的依赖，来避免依赖模块的移除和重新安装。

现在 benchmark 已经有了正确的配置，当其他用户使用并不会再去安装这个模块。

可选的依赖

可选的依赖不是一个项目运行的必需条件，但是可以和正常的依赖一样被安装。唯一的区别是可选的依赖安装失败时，模块会忽略掉其他的依赖继续安装。

这通常用于一些利用原生代码来提升性能的插件。例如，hiredis 是一个使用原生的 C 代码编写的插件，用于提升 redis 模块的性能。但是和它无法适配各个环境，一旦无法安装时，会降级使用 JavaScript 实现的 redis 模块。通常在模块中用于检测依赖是否存在的方法是：

```
try {
  var client = require('hiredis'); // 超级快的方法!
}
catch (e) {
  var client = require('./lib/redis'); // 一般快的方法
}
```

```
module.exports = client;
```

- ❶ 尝试加载可选的模块。
- ❷ 加载可选模块失败时的处理，通常是使用另外一种实现方式来替代可选模块。
- ❸ 对外暴露通用的借口，使用时无须关注是否引用了可选的模块。

我们尝试在 fastfib 模块中支持大数数列的计算。使用一个原生代码实现的 bigint 模块来实现这个功能：

```
npm install bigint --save-optional
```

然后在入口文件对 bigint 进行检测，来判断是否使用迭代方式实现的方法：

```
try {  
  var fastfib = require('./lib/bigiter');  
}  
catch (er) {  
  var fastfib = require('./lib/iter');  
}  
  
module.exports = fastfib;
```

- ❶ 尝试使用 `bignum` 的实现模块。
- ❷ 引用 `bignum` 失败则使用迭代的实现方式。

不幸的是，`bignum` 的实现会慢一点，因为它无法被 V8 编译器优化。如果我们使用了这个可选依赖和它的实现方式，便是违背了要实现最快的斐波那契数列计算的目标。但是这个说明了有时候需要使用可选的依赖关系（例如，你想支持计算尽可能大的斐波那契数列）。

课外作业

`bignum` 实现方式对应的代码和测试这里没有涉及；尝试实现一个使用 `bignum` 模块实现的版本，看一下我们基准测试的结果会如何。

Peer dependencies

Peer dependencies（<http://blog.nodejs.org/2013/02/07/peer-dependencies/>）是最新的依赖使用场景。Peer dependencies 意味着告知使用者在安装该模块时：依赖的模块已经以这个版本安装在项目中，以便模块正常运行。这一类依赖最常见的使用是在插件模块中：

流行的一些拥有插件的模块如下：

- Grunt
- Connect
- winston
- Mongoose

如果我们说需要添加一个 `Connect` 中间件在每一次请求中来计算斐波那契数列；谁都不会这么干的，是吗？为了实现这个想法，我们需要确保编写的 API 在正确的 `Connect` 版本下运行。例如，我们的模块在 `Connect 2` 下才能正常地工作，但无法在 `Connect 1` 和 `3` 中运行。这样我们需要在 `package.json` 文件中添加这个配置项：

```
"peerDependencies": {  
  "connect": "2.x"  
}
```

❶

❶ 只允许模块在已经安装了 Connect 2.x 的情况下安装。

在这一节内容中我们讲述了在 `package.json` 中可以定义的四种依赖类型。如果你想要了解 `^1.0.0` 或者 `2.x` 代表的意义，我们会在下一节中深入讲解，但首先我们需要了解如何更新现有的依赖。

保持最新版本的依赖

维护一个健康的模块，也意味着维持你的依赖是最新版本的。幸运的是，有现成的工具来帮助实现这一点。内置的一个工具是 `npm outdated`，严格地按照 `package.json` 文件中声明的依赖版本来查看是否有新的依赖版本。

因为 `npm install` 提供的是最新的版本，所以我们修改一下 `package.json` 文件来尝试更新 `benchmark` 模块版本：

```
"devDependencies": {  
  "benchmark": "^0.2.0"  
},
```

❶

❶ 把 `benchmark` 回滚到早前的版本。

然后执行 `npm outdated` 来看一下会如何：

```
$ npm outdated  
Package Current Wanted Latest Location  
benchmark 1.0.0 0.2.2 1.0.0 benchmark
```

看起来应该是安装了 1.0.0 的版本，但是根据刚刚修改的 `package.json` 配置，我们希望最新的包版本匹配 `^0.2.0`，那么便是提供了 0.2.2 的版本。我们可以看到最新可用的包版本是 1.0.0。Location 这一项会告诉我们在哪里发现需要更新的依赖。

目录引入的过期依赖

通常我们很乐意看到当前目录的过期依赖，而非子目录中的一些依赖（在一个大项目中会非常多）。`npm outdated --depth 0` 这个命令可以帮你实现这个效果。

如果我们要更新需要的版本，可以运行：

```
npm update benchmark --save-dev
```

这样会安装 0.2.2 版本并且更新 `package.json` 文件中对应的配置为 `^0.2.2`。

再执行一下 npm outdated :

```
$ npm outdated
Package Current Wanted Latest Location
benchmark 0.2.2 0.2.2 1.0.0 benchmark
```

现在已经有需要的对应版本的依赖了。如果希望更新到最新的版本，直接安装最新的版本并且把配置保存到 package.json 中便可以实现：

```
npm install benchmark@latest --save-dev
```

版本标签和范围

注意到使用 @latest 标签来获取一个模块最新发布的版本。npm 同时也支持指定特殊版本和版本范围的功能 (<https://www.npmjs.org/doc/cli/npm-install.html>)。

之前我们已经提及了一部分版本号的内容，但是真的需要一个技巧章节来专门讲解，因为理解版本号的含义以及如何有效地使用是十分重要的。理解语义化的版本号有助于你更好地定义你的模块和依赖的版本。

技巧 110 语义化版本号

如果你对语义化版本号不熟悉，可以看一下 <http://semver.org>。图 13.4 展示了主要的内容。

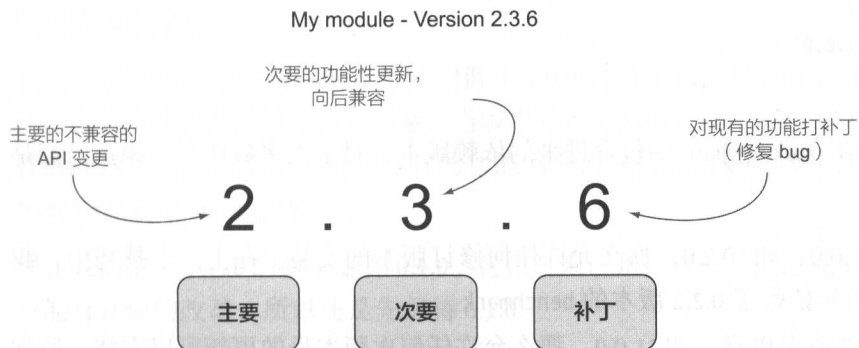


图 13.4 语义化的版本

这里是官方文档的相关描述：¹

¹参见 <http://semver.org/>。

版本格式：主版本号.次版本号.修订号，版本号递增规则如下：

1. 主版本号：当你做了不兼容的 API 修改。
2. 次版本号：当你做了向下兼容的功能性新增。
3. 修订号：当你做了向下兼容的问题修正。

在实践中，这些规则有时候会被忽略或者没有严格地遵循，毕竟，谁也不会强制你的版本号。同时，很多作者在早期阶段很喜欢修改 API，也不会一下子就把版本号设置为 24.0.0。但是语义化起码可以提供给作者或者用户一个线索去了解上一次发布后发生了什么情况。

在这一节内容中我们看一下在 `fastfib` 中如何有效地使用语义化版本号。

问题

希望在模块中有效地使用 `semver`，包括管理依赖。

解决方案

考虑你项目的潜在发展来确定一个安全的升级路径，在你的版本中清晰地表达你的意图。

讨论

现在在项目的开发依赖中有一个依赖的模块，如下：

```
"devDependencies": {  
  "benchmark": "^1.0.0"  
},
```

这个是 `npm` 默认在 `package.json` 中包含进来的依赖版本，对于大多数作者来说这已经是足够了：

- 如果版本低于 1.0.0，如 `^0.2.0`，那么允许任何修订版本的安装。在上一个技巧中，我们可以看到最后是依赖了 0.2.2 版本的 `benchmark`。
- 如果版本是 1.0.0 或者更高，如 `^1.0.0`，那么允许任何次版本号的更新可以安装。通常 1.0.0 是被认为是稳定的版本，并且次版本号的更新不会破坏原有的 API。

这就意味着另外一个用户在安装你模块的依赖时，他们会在你允许的范围内获取最新的依赖版本。例如，如果 `Benchmark.js` 明天发布了 1.1.0 版本，尽管在你的电脑中还是 1.0.0 的版本，但是新的用户则会使用 1.1.0 的依赖版本，因为它还是在你设置的版本范围内。

版本操作符

Node 支持一整套的版本操作符来定义多个版本或者版本范围。你可以在 semver 文档中查看 (<https://www.npmjs.org/doc/misc/semver.html>)。

依赖的版本控制

当编写模块时，使用指定的依赖版本，确保用户在安装模块时所使用的依赖版本，会让整个项目的实现更加有把握。这样做，你可以知道在同等依赖下的测试结果。我们知道项目是在 benchmark 1.0.0 中进行测试的，可以使用以下的命令来锁定当前的整个版本：

```
npm install benchmark --save-dev --save-exact
```

❶ 在 package.json 中保存确切的依赖安装版本。

当然，我们的 package.json 文件已经被更新了，可以看一下：

```
"devDependencies": {  
  "benchmark": "1.0.0"  
},
```

❶ 不带任何其他标示符的确切版本号。

现在我们已经锁定了项目的依赖，可以经常使用 npm outdated 来查看是否有新的版本，并且使用 npm install 和 --save-exact 标识来更新 package.json 配置。

模块的版本控制

我们已经提到，很多模块的作者使用低于 1.0.0 的版本来标识 API 还未完全实现并且可能在次版本变更时频繁更新。通常，当版本号是 1.0.0 时，则代表了该模块是稳定的，尽管 API 接口可能会增加，但是现有的功能不会再进行大调整。这符合 npm 保存一个模块到 package.json 配置中的行为。

现在我们把 fastfib 的版本定在 0.1.0。它相对来说是稳定的，但是在 1.0.0 版本之前我们可能有其他的改动，所以还是先保持 0.1.0。

变更日志

模块的作者可以维护一个变更日志来总结一个新的版本发布时用户应该注意哪些东西，这是非常有用的。变更日志可以参考以下格式：

```
Version 0.5.0?--?2014-04-03  
---  
added; feature x
```



```
removed; feature y [breaking change!]  
updated; feature z  
fixed; bug xx
```

```
Version 0.4.3?--2014-03-25  
---
```

破坏性的变更，尤其是在次级版本的更新中，应该在变更日志中明确声明，以告知用户如何应对版本更新。一些模块的作者喜欢在 `readme` 文档中维护一份变更日志，或者使用一个独立的变更日志文件。

我们已经介绍了一些关于依赖和模块版本化的概念和工具；接下来看一下还能够为模块的使用者提供哪些东西。

13.3 用户体验

在把模块发布给用户使用之前，最好是测试一下能否正常工作。当然，我们已经有了测试套件，所以我们知道我们的逻辑是合理的，但是什么才是一个用户安装模块良好的使用体验？除了 API 之外如何暴露一个可以执行的脚本给用户？可以支持哪些版本的 Node？在这个章节中我们会看一下如何解决这些问题，先从添加一个可执行脚本开始。

技巧 111 添加可执行脚本

当你的模块被安装时想要对外暴露一个可执行的命令？例如 Express，包括了一个可运行的 `express` 命令，你在命令行中可以直接运行来帮助初始化一个新项目：

```
$ npm install express -g  
$ express
```

❶

❶ 全局安装 `express` 模块，来让其在哪里都能够运行。

`npm` 本身就是一个安装模块，带有可执行的命令 `npm`，我们已经在这个章节使用了好多次。

可执行的命令让用户可以以不同的方式来使用模块。在这一节内容中，我们会给 `fastfib` 添加一个可执行的脚本，在 `package.json` 中配置以便跟随模块一起安装。

问题

如何添加一个可执行的脚本。

解决方案

在 `package.json` 中配置，来给一个模块添加命令行工具或者脚本。

讨论

我们已经构建 `fastfib` 模块，但想要暴露一个 `fastfib` 的可运行命令来给我们的终端用户，他们可以在命令行中使用 `fastfib 40` 来打印斐波那契数列的第 40 个数字，应该怎么做？这需要允许模块在命令行中也能像在程序中一般使用。

为了达到这个目的，我们创建一个 `bin` 目录，来保存 `index.js` 文件，里边包括了以下代码：

```
#!/usr/bin/env node
var fastfib = require('..');
var seqNo = Number(process.argv[2]);
```

```
if (isNaN(seqNo)) {
  return console.error('\nInvalid sequence number provided,
  try:\n fastfib 30\n');
}
```

```
console.log(fastfib(seqNo));
```

- 1 告知操作系统使用可运行的 `node` 来执行下边的代码。
- 2 引用 `fastfib` 模块。
- 3 获取数列的长度参数。
- 4 如果无法获取有效的参数值，则提前退出程序并且输出包含相关说明的错误信息。
- 5 输出结果。

现在我们的应用程序已经可以在命令行中运行了，那怎么把它作为 `fastfib` 命令暴露给安装了模块的用户使用？为了实现这个需求，需要更新 `package.json` 文件。在 `main` 配置项下添加以下配置：

```
"main": "index.js",
"bin": {
  "fastfib": "./bin/index.js"
},
```

- 1 给可执行脚本命名为 `fastfib`，运行时便是执行 `./bin/index.js`。

使用 `npm link` 来测试命令

我们可以使用 `npm link` 来测试模块的运行。`link` 命令会给我们的模块创建一个全局的链接，来模拟全局安装模块，如同一个用户在全局下安装模块一样。

我们在 `fastfib` 下运行 `npm link`：

```
$ npm link
/usr/bin/fastfib
-> /usr/lib/node_modules/fastfib/bin/index.js
/usr/lib/node_modules/fastfib
-> /Users/wavded/Dev/fastfib
```

❶ fastfib 命令链接到 ./bin/index.js 脚本。

❷ fastfib 模块链接到我们工作目录下。

这样我们便有了全局的可执行命令，我们尝试一下：

```
$ fastfib 40
102334155
```

因为链接已经生成了，对于模块的任何修改都会在全局环境下生效。我们更新一下 bin/index.js 的最后一行代码来宣布运行的结果：

```
console.log('The result is', fastfib(seqNo));
```

如果再运行 fastfib 一次，可以立马得到更新后的结果：

```
$ fastfib 40
The result is 102334155
```

我们已经给 fastfib 模块添加了一个可执行命令。有一点需要注意，在这一节中谈及的内容都是完整的跨平台兼容的。Windows 没有符号链接和 #! 声明，但是 npm 使用额外的代码包裹了可运行的脚本来让你在使用 npm link 和 npm install 时有着同样的效果。

符号链接是一个很强大的工具，我们会在下一个技巧中谈及更多的内容！

技巧 112 在本地测试模块

使用 npm link 除了可以来测试全局可执行的模块脚本，还可以在任意地方尝试使用我们的模块。假设我们想在另外一个项目中尝试使用这个新的模块，来看看它是否能够正常工作。我们可以不用发布并且安装模块，只需要链接并且在另外一个项目中使用即可。

问题

你希望在发布前尝试使用一下模块，或者是，修改了模块后，重新发布之前，你希望在另外一个项目尝试使用一下。

解决方案

使用 npm link。

讨论

在上一个技巧中我们展示了如何使用 `npm link` 来尝试跑模块的可执行脚本。这意味着在开发过程中可以尝试运行模块的可执行脚本，但现在我们需要的是模拟模块的本地安装而非全局。

我们先创建另外一个项目。这一章开始的时候我们谈及了 Node 的恶梦是一个处理斐波那契数列的 web 服务器，兜了一圈之后，我们来尝试创建一个项目来实现一个 `fastfib` 的 web 服务。

创建一个名为 `fastfibserver` 的新项目，只有一个简单的 `index.js` 文件，里边是如下内容：

```
var fastfib = require('fastfib');  
var http = require('http');  
  
http.createServer(function (req, res) {  
  res.end(fastfib(40));  
}).listen(3000);  
  
console.log('fastfibber running on port 3000');
```

- ① 引用 `fastfib` 模块。
- ② 每一个请求中返回第 40 个斐波那契数列中的数字。

我们已经创建好服务了，但如果执行 `node server`，还是无法正常运行的，因为在这里项目中还未安装 `fastfib` 模块。使用 `npm link` 来实现：

```
$ npm link ../fastfib  
  /usr/bin/fastfib  
  -> /usr/lib/node_modules/fastfib/bin/index.js  
  /usr/lib/node_modules/fastfib  
  -> /Users/wavded/Dev/fastfib  
  /Users/wavded/Projects/Dev/fastfibserver/node_modules/fastfib  
  -> /usr/lib/node_modules/fastfib  
  -> /Users/wavded/Dev/fastfib
```

- ① 传入 `fastfib` 模块的路径。
- ② 首先创建全局的链接。
- ③ 最后在 `fastwebserver` 目录中设置链接。

现在我们启动服务器，是可以成功运行的：

```
$ node server  
fastfibber running on port 3000
```

浏览我们网站的时候可以看到斐波那契数列在第 40 个位置的数字，如图 13.5 所示。

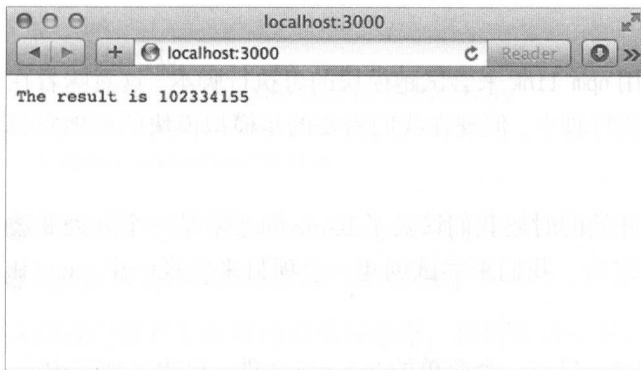


图 13.5 fastfibserver 的输出例子

使用 link 的另外一种方式

在上一个技巧中我们已经在 fastfib 项目中使用 `npm link` 把模块链接成为全局可用的，这样的话，我们也可以在 fastfibserver 项目中运行 `npm link fastfib` 来设置链接。

使用 `npm link` 对于你在不同的项目背景下调试模块有很大的帮助。对边缘情况来说，最好的是在依赖模块的真实项目中去运行才能更好地调试。一旦使用了 `npm link` 来链接模块，任何的更改会立即生效，而无须重新发布和安装。这允许你在模块中一边调试问题一边调整代码。

到目前为止，我们已经定义和实现了模块测试，配置好需要的依赖，锁定了依赖和模块的版本号，添加了一个命令行中可执行的脚本，并且在实际项目中尝试使用我们的模块。下一步来认识一下 `package.json` 文件中的另外一个配置项 `engines`，然后在多个版本的 Node 中来测试我们的模块。

技巧 113 在不同版本 Node 中测试

不幸的是，在实际情况中不是每个人都能够升级到最新版本的 Node。这在公司中需要时间去维护，让代码保持在最新版本的运行环境中。我们需要确认我们的模块能够在哪个版本的 Node 中运行以便让 `npm` 知道哪些用户可以安装和运行，这是非常重要的。

问题

你希望在多个 Node 版本中测试模块，同时希望你的应用只是在特定的 Node 版本中被安装。

解决方案

在多个版本的 Node 中测试，并且在 `package.json` 文件中维护一个正确的 `engines` 配置对象。

讨论

我们第一次初始化 `package.json` 文件时运行 `npm init` 并没有包括一个 `engines` 的配置项，这意味着 `npm` 会在任意版本的 Node 中安装这个模块。一开始看，我们可能觉得这是正常的，因为我们只是运行原生的 JavaScript 代码。但是在没有测试的情况下，我们没法真正确定是否正常。

通常情况下，修订版本的更新（如 Node 0.10.2 到 0.10.3）不应该影响你模块的运行。但是，由于 V8 近期的更新以及 Node APIs 的变动，我们至少应该测试次级版本和主要版本的兼容问题。目前，我们在 Node 的 0.10 分支上模块一切运行良好。我们从这里开始，把下边的内容添加到 `package.json` 文件中去：

```
"homepage": "https://github.com/wavded/fastfib",
"engines": {
  "node": "0.10.x"
}
```

①

① 表示我们的模块可以在 Node 0.10 任意一个补丁版本中运行。

这仅仅只是个开始，看起来我们需要添加对 Node 早期版本的支持，应该怎么测试？

有很多流行方法可以使用：

- 在电脑中安装多个版本的 Node。
- 使用 Travis CI 的多版本 Node 支持（<https://travis-ci.org>）。
- 在对应的环境中使用第三方的多版本测试模块（如 `dnt`——<https://github.com/rvagg/dnt>）。

关于 Node 的版本

在 Node 中，所有奇数次版本号被认为是不稳定的。所以 0.11.0 是 0.12.0 发布前的不稳定版本，等等。你不需要测试现有的不稳定版本，通常，模块的作者只是测试接近完成了的最新的稳定版本。

在这个技巧中，我们会关注在多个版本的 Node 安装，这样可以测试即将发布的 Node 新版本的特性，以及用来测试我们的模块。

我们选择的工具是 `nvm` (<https://github.com/creationix/nvm>，Windows 系统下可以使用 `nvmw`: <https://github.com/jalpnara/nvmw>)。接下来主要是讲解 `nvm` 的使用，和 `nvmw` 的命令有些相似。

安装 `nvm`，在命令行中运行：

```
curl https://raw.githubusercontent.com/creationix/nvm/v0.5.0/install.sh | sh
source ~/.nvm/nvm.sh
```

❶ 我们立即加载用户配置文件，而不需要重新打开 shell 会话，这在将来新版本的 `nvm` 中可能会自动完成。

现在已经安装好了，接下来在 Node 0.8 版本中测试我们的 `fastfib` 模块。首先安装 Node 0.8：

```
$ nvm install 0.8
#####.100.0%
Now using node v0.8.26
```

`nvm` 会拉取 0.8 分支的最新版本。如果需要，我们也可以指定修订版本号，但现在这个已经足够了。注意一下如何使用这个版本，我们可以运行以下命令来验证一下：

```
$ node -v
v0.8.26
```

现在，所有的 Node 和 `npm` 相关的交互都在 0.8.26 版本的一个隔离的环境中进行。如果我们需要安装更多的版本，它们也将在属于自己的一个隔离环境中运行。我们可以使用 `nvm use` 在多个版本中进行切换。例如，如果你想要返回系统原本安装的 Node，你可以运行以下命令：

```
nvm use system
```

如果是切换回 Node 0.8.26 版本则是：

```
nvm use 0.8
```

在 0.8.26 版本中尝试运行我们的测试：

```
$ npm test

> fastfib@0.1.0 test /Users/wavded/Dev/fastfib
> node test && node benchmark

results:
recurse 432 5.48
tail 300770 5.361
iter 1109759 5.428
```

看起来是没问题的！在 `package.json` 更新一下，把 0.8 版本包括进来：

```
"engines": {  
  "node": ">=0.8.0 <0.11.0"  
}
```

❶

❶ 版本 0.8.0 以上，低于版本 0.11.0。

如果我的模块失去了一个特定的 Node 版本的支持会怎样？

这是完全没有问题的。旧版本的 Node 用户会获取到支持对应版本的最新发布的包。

我们已经测试了 0.10 和 0.8 Node 版本，你可以自己尝试一下其他的版本。当完成测试后，别忘记把 Node 版本切换回系统原本的。

现在，我们已经按部就班地把模块准备好了，可以提供给用户使用了，让我们发布吧！

13.4 发布

在我们结束这一章内容之前，我们先来重点看一下如何在 `npm` 上公开发布一个模块或者使用一个内部私有模块。

技巧 114 发布模块

嗯！我们已经在许多个技巧讨论中为我们的模块准备好发布了。我们知道可能还会调整，但是已经可以发布第一个版本，让其他的项目可以使用这个模块。这一节内容会讲述发布的多个方面内容。

问题

你想要公开发布你的模块。

解决方案

如果还未注册 `npm`，先注册，然后使用 `npm publish` 来发布。

讨论

如果这是你第一次发布一个模块，那么需要先在 `npm` 上注册。幸运的是，注册已经简单地不能再简单了。运行以下命令并且依照提示做便可：

```
npm adduser
```

注册完成后，`npm` 会把用户证书保存在 `.npmrc` 文件中。

修改已有的账号信息

`adduser` 命令同样也能用于修改用户账号的信息（除了用户名）以及设置一个现有的账号信息。

完成注册后，发布模块就如同添加一个用户一样简单。但在发布前，我们需要先了解一下发布模块的一些比较好的实践方式。

在发布之前

在发布前最重要的一点是检查一下在技巧 110 中提及的语义化版本：

- 你的版本号是否已经反馈了最后一个更新的内容？如果是第一次发布，那么这个可能无关紧要。
- 你是否为你的这次发布更新了变更日志？尽管这不是必要的，但是对于依赖你项目的用户来说，在更新时能够获得更进一步的信息是非常有用的。

同样地，检查一下你的测试用例是否通过，来避免提交错误代码。

发布到 npm

当你准备好发布时，很简单地，只需要在项目根路径下运行以下命令即可：

```
npm publish
```

npm 会返回发布是成功还是失败。如果成功，会告知已经推送到 npm 源的模块版本。

你可以说是 npm 希望你尽可能轻松地把模块发布出去？

撤销发布

尽管我们希望模块发布一切正常，但有时候我们可能漏了要发布的东西，或者在发布后发现了一些错误。建议不要移除发布的版本（即使提供了这个功能）。原因是依赖这个模块/版本的人们可能再也无法获取相应的内容了。

通常的话，是修复之后，修订版本号加一，再次执行 `npm publish` 来发布。一个更简单的方法是使用如下命令：

```
// make fixes  
$ npm version patch  
v0.1.1  
$ npm publish
```


❶ `npm version` 命令 (<https://www.npmjs.org/doc/cli/npm-version.html>) 会根据传递的参数来更新 `package.json`。这里我们使用 `patch` 来让其增加修订版本号。

`npm` 不会允许你发布一个已经存在的版本，因为这可能会影响已经下载了对应版本的用户。

某些情况下，你真的想阻止用户去使用某一个特定的版本。例如，在版本 0.2.5 以上修复了一个严重的安全漏洞，但你还有一些用户依赖于早前的版本。使用 `npm deprecate` 可以帮助你弃用某些版本。

我们假设在 `fastfib` 的 0.2.5 版本中发现了一个致命的错误，希望警告那些正在使用这个模块的用户。我们可以这么做：

```
npm deprecate fastfib@<= 0.2.5  
"major security issue was fixed in v0.2.6"
```

❶

❶ `npm` 会显示弃用的模块名称，紧跟着的是影响的版本范围。

现在如果任意一个用户安装 `fastfib 0.2.5` 或者更低版本，他们会收到 `npm` 发出的相应的警告。

技巧 115 使用私有模块

虽然开源是一件十分有趣的事，同时可以获得一个协作维护的环境，但有时候你希望你的项目是私有的。在为某些客户工作时这种情况确实是可能发生的。同样地，首先在内部开发一个模块，然后再决定是否公开发布也是很常见的情况。`npm` 可以让你的模块维持一个私有的状态。在这一节内容中，我们会讲述把模块配置为私有的，以及在项目中引用私有模块的方法。

问题

你想要让模块私有化，在内部中使用。

解决方案

在 `package.json` 中配置 `private`，并且在内部分享。

讨论

我们想要 `fastfib` 只是在内部使用。为了确保它不会因为意外而发布出去，需要在 `package.json` 中添加以下配置：

```
"private": true
```

这会让 npm 在使用 npm publish 时拒绝发布。

这个配置对于特定客户的项目非常适用。但是如果你有一些内部模块需要在开发团队中共享怎么办？下边有几个不同的选择方案。

使用 Git 来共享私有模块

npm 支持多种方法使用最小的开销来共享你的内部模块。如果你使用 Git 代码仓库，npm 使得这一切都变得非常简单。

就拿 Github 来作为例子（它也可以是任意一个 Git 远程仓库）。我们假设有一个私有的代码仓库：

```
git@github.com:mycompany/fastfib.git
```

我们可以使用 npm install 把它作为依赖引用进来（或者直接修改 package.json 文件）：

```
npm install git+ssh://git@github.com:mycompany/fastfib.git --save
```

看起来真棒！默认会拉取 master 分支上的内容。如果需要指定某一个特定的提交记录点（tag、branch 或者 SHA-1——<http://git-scm/book/en/Git-Internals-Git-Objects>），也是可以实现的！可以看 package.json 配置的一些例子：

```
"dependencies": {  
  "a": "git+ssh://git@github.com:mycompany/a.git#0.1.0",  
  "b": "git+ssh://git@github.com:mycompany/b.git#develop",  
  "c": "git+ssh://git@github.com:mycompany/c.git#dacc525c"  
}
```

❶
❷
❸

- ❶ 指定 Git tag。
- ❷ 指定分支名称。
- ❸ 指定提交的 SHA-1（通常不会使用完整的 SHA-1）。

引用公共的代码参考

你可能已经猜到了，我们也可以使用公共的 Git 代码仓库。当你需要一个还未在 npm 中发布的模块特性或者 bug 修复时，这会非常有用。更多的例子可以参考 package.json 文档（<https://www.npmjs.org/doc/json.html#Git-URLs-as-Dependencies>）。

利用 URL 来共享私有模块

如果你没有使用 Git 或者更希望使用自己的构建系统来输出包，你可以提供一个 URL 给 npm 以便其找到对应的包。要打包你的模块，可以使用 tar 命令：

```
tar -czf fastfib.tar.gz fastfib
```

❶

❶ 我们使用 `tar` 来给 `fastlib` 目录打包，使用 `gzip` 来进行压缩，然后用文件 `fastfib.tar.gz` 存储起来。

然后，我们可以把这个文件放到 web 服务器上，然后使用以下方法来安装：

```
npm install http://internal-server.com/fastfib.tar.gz --save
```

关于公共包的使用

尽管通常都不会使用，但是压缩包也是可以使用公共端的服务的。正常情况下，`npm` 会更加方便易用。

使用私有的 `npm` 源来共享私有模块

另外一个私有仓库的选择便是使用自己私有的 `npm` 源，并且使用 `npm publish` 来推送到对应的私有源上。要使用完整的 `npm` 功能，需要安装一个较新版本的 `CouchDB`，而这也依赖于 `Erlang`。

由于这个使用涉及很多依赖操作系统的麻烦的事情，在这里我们并没有提供一个例子。希望这个使用过程很快可以得到简化。如果你想尝试一下，可以参考 `npm-registry-couchapp` 项目（<https://github.com/npm/npm-registry-couchapp>）。

13.5 总结

第三方模块便是创新所在的地方。`npm` 让这一切变得更加简单和有趣！随着社会化编码，如 `Github` 等网站的兴起，模块开发的协作也变得更加容易。在本章中，我们可以看到模块开发中的各个方面，让我们来总结一下这些经验教训。

当开始一个模块的开发时，需要考虑以下问题：

- 确定你的想法。你可以使用一句话来概括这个模块的作用吗？
- 检查一下想法可行性。是否已经有另外一个模块实现了你所需要的？可以使用 `npm search` 来搜索或者在 `npmjs.org` 寻找。

当你准备落实一个想法时，便可以开始了。首先从可能用到的简单 `API` 开始。开始编写实现和测试，并且安装在过程中需要的依赖。

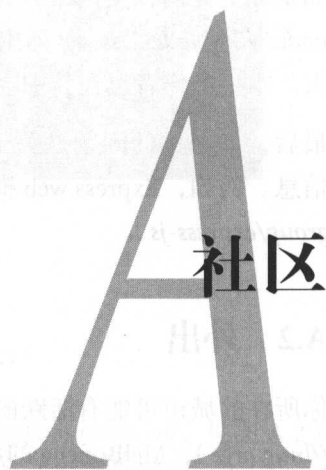
在想法实现时（或者实现过程中），以下的事情是你需要考虑的：

- 你已经初始化项目的 `package.json` 文件了？运行 `npm init` 来获取基本的脚手架，用于表示当前项目的状态。
- 处理好你的依赖，是否有一些是可以选的，或者开发专用的？确保它们都已经在 `package.json` 有相应的声明。
- 检查 `package.json` 中版本范围。`package.json` 指定的版本范围是否是你可以信任的。使用 `npm outdated` 来检查是否需要更新。
- 你的代码需要运行在哪一个版本的 Node。可以使用 `nvm` 或者集成构建工具，类似 Travis CI 来进行测试。在 `package.json` 指定需要的版本范围。
- 使用 `npm link` 在另外一个项目使用自己开发的模块。

当你准备好发布的时候，很简单，使用 `npm publish` 就可以了。如果使用语义化的版本号，可以考虑为你的用户维护一个变更日志，这样你的用户能够对版本的调整有一个更好的了解。

在这里再一次给本书进行总结！我们希望你能从中掌握 Node 的基础，能够了解在现实世界的场景中如何去应用，以及在基础上开发属于自己的 Node 模块（这正是我们希望在 npm 上所看到的！）。

现在迅速成长的 Node 社区可以帮助你更进一步地提高水平。请注意查看大部分社区的相关附件。如果你对我们有意见或者建议，可以访问谷歌的 `#nodejsinpractice` 讨论组（<https://groups.google.com/forum/#!forum/nodejsinpractice>）。非常感谢阅读本书！



这一部分内容会帮助你在 Node 社区中好好进步。编程社区可以帮你找到各种在文档中没有直接说明的问题的答案。你可以更有效地和志同道合的人一起学习，无论是线上或者亲身见面。

A.1 提问

有时候你只是想要知道一些看起来好像很容易的事情怎么做，但其实它并不简单。有时候你觉得你发现了 Node 一个重要的 Bug。无论是什么情况，当你需要帮助且 Node API 文档无法满足的时候，有一些你可以使用的官方的渠道。

第一个是 Node 的邮件列表，即 nodejs 的谷歌群组（<http://groups.google.com/group/nodejs>）。你可以使用 Google 的 web 接口来订阅。web 接口允许搜索所有的文章，所以你可以看一下是否有其他人已经问了相同的问题。

这个群组已经有一些突出贡献的社区成员，包括 Isaac Schlueter、Mikeal Rogers 和 Tim Caswell，所以这个群组是获得帮助和了解 Node 的好地方。

还有一个正式的 IRC 聊天室：irc.freenode.net 的 #node.js。它极其忙碌，请把很多信息都准备好。#node.js 经常有重要信息的讨论，所以请耐心，总会有回报。

如果你是 Stack Exchange 网络的粉丝，你可以使用 `node.js` 标签来提交问题（<http://stackoverflow.com/questions/tagged/node.js>）。

如果你更倾向社交网络，在 Node wiki 列表的 Node 用户群组（<https://github.com/joyent/node/wiki/Node-Users>）列出了几百个在各个开发者时区的 Twitter 账号，你可以找人聊天。本书的作者也在其中。

最后，如果你的问题是关于某一个特定的模块，你可以查看那个模块文档来找到社区信息。例如，Express web 框架有它自己的 `express-js` 谷歌群组（<https://groups.google.com/group/express-js>）。

A.2 外出

你所在的城市可能有活跃的 Node 会议群组。例如在 London 的 Node.js 用户群组（<http://lnug.org/>），Melbourne 的 Node.js 会议组（<http://www.meetup.com/MelbNodeJS/>），还有在 California Mountain View 的 BayNode（<http://meetup.com/BayNode/>）。

还有主要的 Node 大会，包括 NodeConf（<http://nodeconf.com/>）和 NodeConf EU（<http://nodeconfeu.com/>）。

为了帮助你找到更多满足需要的团体和会议，Node.js Meatspace 页面（<https://github.com/knode/node-meatspace>）会经常更新。当然你也可以搜索 meetup.com。

A.3 阅读

如果你正在寻找一些读物，你会发现一些不错的公共社区。reddit.com/r/node 会收集一些好文章，在 Medium 也有一些好收藏，包括 medium.com/node-js-javascript。

留意 Node 开发者也有你可以浏览的博客。Isaac Z. Schlueter（<http://blog.izs.me/>）、James Halliday（<http://substack.net/>，见图 A.1），以及 Tim Caswell 等都有个人博客编写和 Node 相关的东西。Tim 的 howtonode.org 有很多适合初学者的资料，但也会帮你跟踪新的发展。

还有一些商业博客会有一些来自优秀 Node 开发者的贡献。Joyent 的博客 joyent.com/blog 有很多关于 Node 发布的有趣文章。还有 StrongLoop 的博客，“In the Loo”，strongloop.com/strongblog。

Nodejitsu 的博客 blog.nodejitsu.com 有一些关于发布的建议，也有一些特别模块的作者谈论他们的作品。

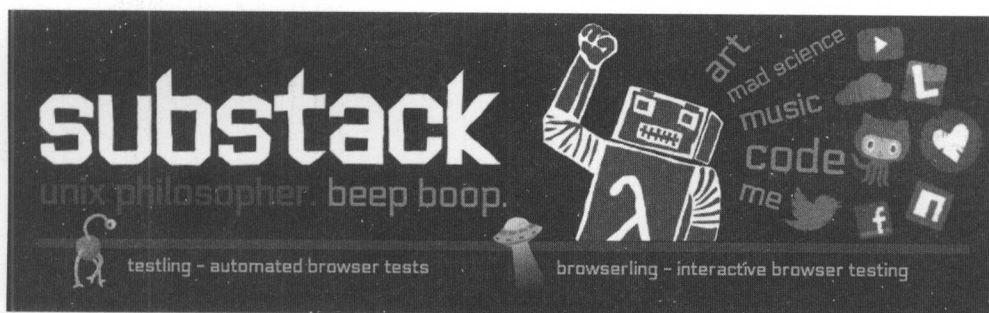


图 A.1 James Halliday 关于 Node 和测试的博客

A.4 为了社区，接受社区的培训

有一个有趣的关于 Node 教学的项目 NodeSchool (<http://nodeschool.io/zh-cn/>，见图 A.2)。你可以自己学习课程，也有社区面对面的培训活动。NodeSchool 提供材料来设立培训项目，这让他们在世界各地迅速膨胀。这个网站有更多详细的内容。

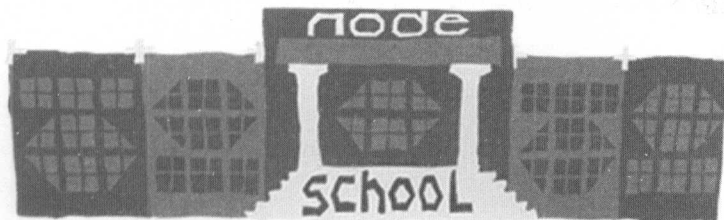


图 A.2 在 NodeSchool 学习 Node

A.5 推销你的开源项目

如果你要参与 Node 社区，最好的方式就是分享你的工作成果。但是 npm 现在是如此流行，很难让你的模块得到关注。

要真正发挥影响力，你应该考虑开源项目的营销方法。如果 Node 的博主们有联系方式或者 Twitter，告诉他们你正在做什么有益而无害。只要你有礼貌，提供好容易理解的一个工作背景，那么可以帮你获得反馈以及提高你的能力。

你决定在下一个项目中使用Node.js，你需要在生产环境中使用Node。这本书可以让你如同有Alex Young和Marc Harter两位专家在你身旁来帮助你应对那些日常的挑战一般！

Node.js实战中有115个已经通过测试的例子和真实有用的技术来帮助任何Node应用跑得更好。本书使用常见的提出问题/解决问题的模式来讲解，书中这些带有丰富经验加持的技术内容，囊括了很多重要的主题，如基于事件的编程、流、集成外部应用和发布等。带有丰富注释的代码会让例子更加容易理解。所有的技术内容按照逻辑划分成多个部分，可以帮助你快速找到想要的东西。

本书包含

- 从基础到进阶的常见使用例子
- 设计和编写模块
- 测试和调试Node应用
- Node和现有的系统集成

本书适合于对JavaScript和Node.js基础有一定了解的读者。

Marc Harter致力于构建大型项目，包括高可用的实时应用，流式接口和其他数据密集型系统。**Alex Young**是一位经验丰富的JavaScript开发者，定期在DailyJS博客发表文章。

“一次Node.js的深度探索。”

——本书作者Ben Noordhuis
StrongLoop公司联合创始人

“充满真实案例的Node.js不传之秘！”

——Kevin Baister
1KB软件解决方案公司

“写给服务端JavaScript工程师的必备实战手册。”

——Gregor Zurowski
苏富比拍卖行

“指导问题排查、调试和解决的实用技术资料。”

——Michael Piscatello
MBP有限责任公司



责任编辑：张春雨
封面设计：李玲

上架建议：前端>JavaScript

ISBN 978-7-121-30402-6



9 787121 304026 >

定价：109.90元